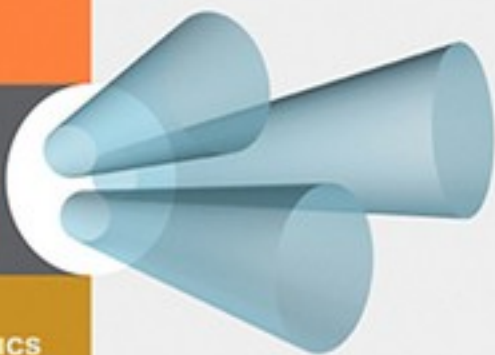


DAVID H. EBERLY

SERIES IN INTERACTIVE 3D TECHNOLOGY

3D GAME ENGINE **DESIGN**

A PRACTICAL APPROACH TO
REAL-TIME COMPUTER GRAPHICS
SECOND EDITION



3D GAME ENGINE DESIGN

A Practical Approach to Real-Time Computer Graphics

SECOND EDITION

DAVID H. EBERLY

Geometric Tools, Inc.



ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier



MORGAN KAUFMANN PUBLISHERS

Senior Editor Tim Cox
Publishing Services Manager George Morrison
Senior Production Editor Brandy Lilly
Project Management Elisabeth Beller
Cover Design Chen Design Associates, San Francisco
Text Design Rebecca Evans
Composition Windfall Software, using $\text{ZzT}_{\text{E}}\text{X}$
Technical Illustration Dartmouth Publishing
Copyeditor Yonie Overton
Proofreader Jennifer McClain
Indexer Steve Rath
Interior and Cover Printer Hing Yip Printers, Ltd.

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

© 2007 by Elsevier Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, scanning, or otherwise—without prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, e-mail: permissions@elsevier.com. You may also complete your request online via the Elsevier homepage (<http://elsevier.com>), by selecting "Support & Contact" then "Copyright and Permission" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data: Application submitted

ISBN 13: 978-0-12-229063-3

ISBN 10: 0-12-229063-1

For information on all Morgan Kaufmann publications, visit our Web site at www.mkp.com or www.books.elsevier.com.

Printed in the United States of America

10 09 08 07 06 5 4 3 2 1



TRADEMARKS

The following trademarks, mentioned in this book and the accompanying CD-ROM, are the property of the following organizations:

- AltiVec is a trademark of Freescale Semiconductor.
- DirectX, Direct3D, Visual C++, Windows, Xbox, and Xbox 360 are trademarks of Microsoft Corporation.
- GameCube is a trademark of Nintendo.
- GeForce, Riva TNT, and the Cg Language are trademarks of NVIDIA Corporation.
- Java 3D is a trademark of Sun Microsystems.
- Macintosh is a trademark of Apple Corporation.
- Morrowind and The Elder Scrolls are trademarks of Bethesda Softworks, LLC.
- NetImmerse and Gamebryo are trademarks of Emergent Game Technologies.
- OpenGL is a trademark of Silicon Graphics, Inc.
- Pentium and Streaming SIMD Extensions (SSE) are trademarks of Intel Corporation.
- PhysX is a trademark of Ageia Technologies, Inc.
- Playstation 2 and Playstation 3 are trademarks of Sony Corporation.
- PowerPC is a trademark of IBM.
- Prince of Persia 3D is a trademark of Brøderbund Software, Inc.
- 3DNow! is a trademark of Advanced Micro Devices.
- 3D Studio Max is a trademark of Autodesk, Inc.

ABOUT THE AUTHOR

Dave Eberly is the president of Geometric Tools, Inc. (www.geometrictools.com), a company that specializes in software development for computer graphics, image analysis, and numerical methods. Previously, he was the director of engineering at Numerical Design Ltd. (NDL), the company responsible for the real-time 3D game engine, NetImmerse. He also worked for NDL on Gamebryo, which was the next-generation engine after NetImmerse. His background includes a BA degree in mathematics from Bloomsburg University, MS and PhD degrees in mathematics from the University of Colorado at Boulder, and MS and PhD degrees in computer science from the University of North Carolina at Chapel Hill. He is the author of *Game Physics* (2004) and *3D Game Engine Architecture* (2005) and coauthor with Philip Schneider of *Geometric Tools for Computer Graphics* (2003), all published by Morgan Kaufmann. As a mathematician, Dave did research in the mathematics of combustion, signal and image processing, and length-biased distributions in statistics. He was an associate professor at the University of Texas at San Antonio with an adjunct appointment in radiology at the U.T. Health Science Center at San Antonio. In 1991, he gave up his tenured position to retrain in computer science at the University of North Carolina. After graduating in 1994, he remained for one year as a research associate professor in computer science with a joint appointment in the Department of Neurosurgery, working in medical image analysis. His next stop was the SAS Institute, working for a year on SAS/Insight, a statistical graphics package. Finally, deciding that computer graphics and geometry were his real calling, Dave went to work for NDL (which is now Emergent Game Technologies), then to Magic Software, Inc., which later became Geometric Tools, Inc. Dave's participation in the newsgroup *comp.graphics.algorithms* and his desire to make 3D graphics technology available to all are what has led to the creation of his company's Web site and his books.

CONTENTS

Preface.....	xxi

1 Introduction.....	1

- 1.1 The Evolution of Graphics Hardware and Games
- 1.2 The Evolution of This Book and Its Software
- 1.3 A Summary of the Chapters

2 The Graphics System.....	7

- 2.1 The Foundation
 - 2.1.1 Coordinate Systems
 - 2.1.2 Handedness and Cross Products
 - 2.1.3 Points and Vectors
- 2.2 Transformations
 - 2.2.1 Linear Transformations
 - 2.2.2 Affine Transformations

- 2.2.3 Projective Transformations
- 2.2.4 Properties of Perspective Projection
- 2.2.5 Homogeneous Points and Matrices
- 2.3 Cameras
 - 2.3.1 The Perspective Camera Model
 - 2.3.2 Model or Object Space
 - 2.3.3 World Space
 - 2.3.4 View, Camera, or Eye Space
 - 2.3.5 Clip, Projection, or Homogeneous Space
 - 2.3.6 Window Space
 - 2.3.7 Putting Them All Together
- 2.4 Culling and Clipping
 - 2.4.1 Object Culling
 - 2.4.2 Back Face Culling
 - 2.4.3 Clipping to the View Frustum
- 2.5 Rasterizing
 - 2.5.1 Line Segments
 - 2.5.2 Circles
 - 2.5.3 Ellipses
 - 2.5.4 Triangles
- 2.6 Vertex Attributes
 - 2.6.1 Colors
 - 2.6.2 Lighting and Materials
 - 2.6.3 Textures

- 2.6.4 Transparency and Opacity
- 2.6.5 Fog
- 2.6.6 And Many More
- 2.6.7 Rasterizing Attributes
- 2.7 Issues of Software, Hardware, and APIs
 - 2.7.1 A General Discussion
 - 2.7.2 Portability versus Performance
- 2.8 API Conventions
 - 2.8.1 Matrix Representation and Storage
 - 2.8.2 Matrix Composition
 - 2.8.3 View Matrices
 - 2.8.4 Projection Matrices
 - 2.8.5 Window Handedness
 - 2.8.6 Rotations
 - 2.8.7 Fast Computations using the Graphics API

3 Renderers.....147

- 3.1 Software Rendering
- 3.2 Hardware Rendering
- 3.3 The Fixed-Function Pipeline
- 3.4 Vertex and Pixel Shaders
- 3.5 An Abstract Rendering API

4 Special Effects Using Shaders.....217

- 4.1 Vertex Colors
- 4.2 Lighting and Materials
- 4.3 Textures
- 4.4 Multitextures
- 4.5 Bump Maps
- 4.6 Gloss Maps
- 4.7 Sphere Maps
- 4.8 Cube Maps
- 4.9 Refraction
- 4.10 Planar Reflection
- 4.11 Planar Shadows
- 4.12 Projected Textures
- 4.13 Shadow Maps
- 4.14 Volumetric Fog
- 4.15 Skinning
- 4.16 Miscellaneous
 - 4.16.1 Iridescence
 - 4.16.2 Water Effects
 - 4.16.3 Volumetric Textures

5 Scene Graphs.....315

- 5.1 The Need for High-Level Data Management
- 5.2 The Need for Low-Level Data Structures
- 5.3 Geometric State
 - 5.3.1 Vertices and Vertex Attributes
 - 5.3.2 Transformations
 - 5.3.3 Bounding Volumes
- 5.4 Render State
 - 5.4.1 Global State
 - 5.4.2 Lights
 - 5.4.3 Effects
- 5.5 The Update Pass
 - 5.5.1 Geometric State Updates
 - 5.5.2 Render State Updates
- 5.6 The Culling Pass
 - 5.6.1 Hierarchical Culling
 - 5.6.2 Sorted Culling
- 5.7 The Drawing Pass
 - 5.7.1 Single-Pass Drawing
 - 5.7.2 Single Effect, Multipass Drawing
 - 5.7.3 Multiple Effect, Multipass Drawing
 - 5.7.4 Caching Data on the Graphics Hardware
 - 5.7.5 Sorting to Reduce State Changes

- 5.8 Scene Graph Design Issues
 - 5.8.1 Organization Based on Geometric State
 - 5.8.2 Organization Based on Render State
 - 5.8.3 Scene Graph Operations and Threading
 - 5.8.4 The Producer-Consumer Model

6 Scene Graph Compilers.....353

- 6.1 The Need for Platform-Specific Optimization
- 6.2 The Need for Reducing Memory Fragmentation
- 6.3 A Scene Graph as a Dynamic Expression
- 6.4 Compilation from High-Level to Low-Level Data
- 6.5 Control of Compilation via Node Tags

7 Memory Management.....377

- 7.1 Memory Budgets for Game Consoles
- 7.2 General Concepts for Memory Management
 - 7.2.1 Allocation, Deallocation, and Fragmentation
 - 7.2.2 Sequential-Fit Methods
 - 7.2.3 Buddy-System Methods
 - 7.2.4 Segregated-Storage Methods

- 7.3 Design Choices
 - 7.3.1 Memory Utilization
 - 7.3.2 Fast Allocation and Deallocation

8 Controller-Based Animation.....389

- 8.1 Vertex Morphing
- 8.2 Keyframe Animation
- 8.3 Inverse Kinematics
- 8.4 Skin and Bones
- 8.5 Particle Systems

9 Spatial Sorting.....507

- 9.1 Spatial Partitioning
 - 9.1.1 Quadtrees and Octrees
 - 9.1.2 BSP Trees
 - 9.1.3 User-Defined Maps
- 9.2 Node-Based Sorting
- 9.3 Portals
- 9.4 Occlusion Culling

10 Level of Detail.....529

- 10.1 Discrete Level of Detail
 - 10.1.1 Sprites and Billboards
 - 10.1.2 Model Switching
- 10.2 Continuous Level of Detail
 - 10.2.1 General Concepts
 - 10.2.2 Application to Regular Meshes
 - 10.2.3 Application to General Meshes
- 10.3 Infinite Level of Detail
 - 10.3.1 General Concepts
 - 10.3.2 Application to Parametric Curves
 - 10.3.3 Application to Parametric Surfaces

11 Terrain.....541

- 11.1 Data Representations
- 11.2 Level of Detail for Height Fields
- 11.3 Terrain Pages and Memory Management

12 Collision Detection.....573

- 12.1 Static Line-Object Intersections
- 12.2 Static Object-Object Intersections
- 12.3 Dynamic Line-Object Intersections
 - 12.3.1 Distance-Based Approach
 - 12.3.2 Intersection-Based Approach
- 12.4 Dynamic Object-Object Intersections
 - 12.4.1 Distance-Based Approach
 - 12.4.2 Intersection-Based Approach
- 12.5 Path Finding to Avoid Collisions

13 Physics.....609

- 13.1 Basic Concepts
- 13.2 Particle Systems
- 13.3 Mass-Spring Systems
- 13.4 Deformable Bodies
- 13.5 Rigid Bodies

14 Object-Oriented Infrastructure.....639

- 14.1 Object-Oriented Software Construction
- 14.2 Style, Naming Conventions, and Namespaces

- 14.3 Run-Time Type Information
- 14.4 Templates
- 14.5 Shared Objects and Reference Counting
- 14.6 Streaming
- 14.7 Startup and Shutdown
- 14.8 An Application Layer

15 Mathematical Topics.....681

- 15.1 Standard Objects
- 15.2 Curves
- 15.3 Surfaces
- 15.4 Distance Algorithms
- 15.5 Intersection Algorithms
- 15.6 Numerical Algorithms
- 15.7 All About Rotations
 - 15.7.1 Rotation Matrices
 - 15.7.2 Quaternions
 - 15.7.3 Euler Angles
 - 15.7.4 Performance Issues
- 15.8 The Curse of Nonuniform Scaling

16 Numerical Method.....719

- 16.1 Systems of Equations
- 16.2 Eigensystems
- 16.3 Least-Squares Fitting
- 16.4 Minimization
- 16.5 Root Finding
- 16.6 Integration
- 16.7 Differential Equations
- 16.8 Fast Function Evaluation

17 Rotations.....759

- 17.1 Rotation Matrices
- 17.2 Quaternions
- 17.3 Euler Angles
- 17.4 Performance Issues
- 17.5 The Curse of Nonuniform Scaling

18 Object-Oriented Infrastructure.....783

- 18.1 Object-Oriented Software Construction
- 18.2 Style, Naming Conventions, and Namespaces
- 18.3 Run-Time Type Information
- 18.4 Templates
- 18.5 Shared Objects and Reference Counting

- 18.6 Streaming
- 18.7 Names and Unique Identifiers
- 18.8 Initialization and Termination
- 18.9 An Application Layer

19 Memory Management.....873

- 19.1 Memory Budgets for Game Consoles
- 19.2 Leak Detection and Collecting Statistics
- 19.3 General Memory Management Concepts

20 Special Effects Using Shaders.....897

- 20.1 Vertex Colors
- 20.2 Lighting and Materials
- 20.3 Textures
- 20.4 Multitextures
- 20.5 Bump Maps
- 20.6 Gloss Maps
- 20.7 Sphere Maps
- 20.8 Cube Maps
- 20.9 Refraction
- 20.10 Planar Reflection
- 20.11 Planar Shadows

- 20.12 Projected Textures
- 20.13 Shadow Maps
- 20.14 Volumetric Fog
- 20.15 Skinning
- 20.16 Iridescence
- 20.17 Water Effects

Bibliography

Index

PREFACE

The first edition of *3D Game Engine Design* appeared in print over six years ago (September 2000). At that time, shader programming did not exist on consumer graphics hardware. All rendering was performed using the fixed-function pipeline, which consisted of setting render states in order to control how the geometric data was affected by the drawing pass.

The first edition contained a CDROM with the source code for Wild Magic Version 0.1, which included 1,015 source files and 17 sample applications, for a total of 101,293 lines of code. The distribution contained support only for computers running the Microsoft Windows operating system; the renderer was built on top of OpenGL; and project files were provided for Microsoft Visual C++ 6. Over the years, the source code evolved to Wild Magic Version 3.9, which contained additional support for Linux and Macintosh platforms, had OpenGL and Direct3D renderers, and included some support for shader programming. However, the design of the engine was still based on a fixed-function pipeline. The distribution also included support for multiple versions of Microsoft's compilers, support for other compilers on the various platforms, and contained some tools such as importers and exporters for processing of art assets.

This is the second edition of *3D Game Engine Design*. It is much enhanced, describing the foundations for shader programming and how an engine can support it. The second edition is about twice the size of the first. The majority of the increase is due to a more detailed description of all aspects of the graphics system, particularly about how shaders fit into the geometric pipeline. The material on scene graphs and their management is also greatly expanded. The second edition has more figures and less emphasis on the mathematical aspects of an engine.

The second edition contains a CDROM with the source code for Wild Magic Version 4.0, which includes 1,587 source files and 105 sample applications, for a total of 249,860 lines of code. The Windows, Linux, and Macintosh platforms are still supported, using OpenGL renderers. The Windows platform also has a Direct3D renderer whose performance is comparable to that of the OpenGL renderer. Multiple versions of Microsoft's C++ compilers are supported—versions 6, 7.0, 7.1, and 8.0 (Professional and Express Editions). The MINGW compiler and MSYS environment are also supported on the Windows platform. The Linux platform uses the g++ compiler, and the Macintosh platform uses Apple's Xcode tools.

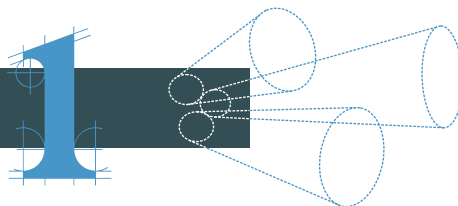
The graphics system of Wild Magic Version 4.0 is fully based on shader programming and relies on NVIDIA's Cg programming language. The precompiled shader programs were created using the arbvp1 and arbfp1 profiles for OpenGL and using the vs_2_0 and ps_2_0 profiles for Direct3D, so your graphics hardware must

support these in order to run the sample applications. If your graphics hardware supports only lesser profiles such as vs_1_1 and ps_1_1, you must recompile the shader programs with these profiles and use the outputs instead of what is shipped on the CDROM. The distribution also contains a fully featured, shader-based software renderer to illustrate all aspects of the geometric pipeline, not just the vertex and pixel shader components.

The replacement of the fixed-function approach by a shader-based approach has made Wild Magic Version 4 a much more powerful graphics engine for use in all graphics applications, not just in games. Much effort went into making the engine easier to use and to extend, and into improving the performance of the renderers. I hope you enjoy this new manifestation of Wild Magic!

A book is never just the product of the author alone. Many people were involved in making this book as good as it possibly can be. Thanks to the reviewers for providing valuable and insightful feedback about the first edition regarding how to improve it for a second edition. A special thanks goes to Marc Olano (University of Maryland, Baltimore County) for taking the time to provide me with detailed comments based on his experience using the first edition as a textbook. Thanks to Elisabeth Beller, the production editor and project manager for all of my Morgan Kaufmann Publisher books, for assembling yet another fine group of people who have the superb ability to take my unattractive book drafts and make them look really good. And, as always, thanks to my editor Tim Cox for his patience and help in producing yet another book for Morgan Kaufmann Publishers.

CHAPTER



INTRODUCTION

I have no fault to find with those who teach geometry. That science is the only one which has not produced sects; it is founded on analysis and on synthesis and on the calculus; it does not occupy itself with probable truth; moreover it has the same method in every country.

— Frederick the Great

1.1 THE EVOLUTION OF GRAPHICS HARDWARE AND GAMES

The first edition of *3D Game Engine Design* was written in the late 1990s when 3dfx Voodoo cards were in style and the NVIDIA Riva TNT cards had just arrived. The book was written based on the state of graphics at that time. Six years have passed between that edition and this, the second edition. Graphics hardware has changed dramatically. So have games. The hardware has extremely powerful graphics processing units (GPUs), lots of video memory, and the option of programming it via shader programs. (These did not exist on consumer cards when I wrote the first edition.) Games have evolved also, having much richer (and much more) content and using more than graphics. We now have physics engines and more recently physics processors (PhysX from Ageia).

The Sony Playstation 2 was not quite released when I started writing the first edition. We've also seen Microsoft's Xbox arrive on the scene, as well as the Nintendo GameCube. These days we have Microsoft's Xbox 360 with multiple processors, and the Sony Playstation 3 is soon to follow with the Cell architecture. Smaller game-playing devices are available. Mobile phones with video screens are also quite popular.

With all this evolution, the first edition of the book has shown its age regarding the discussion of real-time graphics. The time is right for the second edition, so here it is.

1.2 THE EVOLUTION OF THIS BOOK AND ITS SOFTWARE

In the late 1990s when I conceived the idea of writing a book on real-time graphics as used in games, I was employed by Numerical Design, Ltd. (now Emergent Game Technologies) designing and developing NetImmerse (now Gamebryo). At that time the term *game engine* really did refer to the graphics portion of the system. Companies considering using NetImmerse wanted the real-time graphics in order to free up the computer processing unit (CPU) for use by other systems they themselves were used to building: the game logic, the game artificial intelligence (AI), rudimentary collision and physics, networking, and other components. The first edition of *3D Game Engine Design* is effectively a detailed summary of what went into building NetImmerse.

Over the years I have received some criticism for using “game engine” in the title when the book is mainly about graphics. Since that time, the term *game engine* has come to mean a collection of engines—for graphics, physics, AI, networking, scripting, and you name it. It is not feasible to write a book in a reasonable amount of time with sufficient depth to cover all these topics, nor do I intend to write such a massive tome. To address the criticism about the book title, I could have changed the title itself. However, I have chosen to keep the original title—the book is known now by its name, for better or for worse. The second edition includes some discussion about physics and some discussion about an application layer and how the engines must interact, but probably this is not enough to discourage the criticism about the title. So be it.

The first edition appeared in print in September 2000. It is now six years later and the book remains popular in many circles. The algorithmic aspects are still relevant, the scene graph management still applies, but the material on rendering is clearly out of date. That material was essentially about the *fixed-function pipeline* view of a graphics system. The evolution of graphics hardware to support a *shader-based pipeline* has been rapid, now allowing us to concentrate on the special effects themselves (via shader programming) rather than trying to figure out how to call the correct set of state-enabling functions in the fixed-function pipeline to obtain a desired effect.

The second edition of the book now focuses on the design of the scene graph management system and its associated rendering layer. Most of the algorithmic concepts have not changed regarding specialized scene graph classes such as the controller classes, the sorting classes, or level-of-detail classes. Core classes such as the spatial, geometry, and node classes have changed to meet the needs of a shader-based system. The current scene graph management system is much more powerful, flexible, and efficient than its predecessors. The shader effect system is integrated with the scene graph management so that you may do single-pass drawing, multipass drawing with a single effect, or even drawing with multiple effects. I have paid much attention to hiding as many details as possible from the application developer, relying on well-designed and automated subsystems to do the work that earlier versions of my scene graph management system forced the developer to do.

One characteristic of my books that I believe sets them apart from other technical books is the inclusion of large source code libraries, a lot of sample applications that actually compile and run, and support for multiple platforms (PC, Mac, Linux/Unix, and various compilers on each platform). What you have purchased is a book *and* a software product to illustrate what is described in the book. The sample source code that ships with many books is not carefully planned, lacks quality control, is not multiplatform, and usually is not maintained by the book authors. I am interested in carefully designed and planned code. I believe in quality source code. I maintain the source code on a regular basis, so I encourage people to send email about problems they encounter, both with the source code and in the book material. The Geometric Tools website lists all the updates to the software, including bug fixes as well as new features, and the site has pages for book corrections.

The first edition of this book shipped with Wild Magic version 0.1. The book had two additional printings in its first edition, one shipping with Wild Magic version 0.2 and one shipping with Wild Magic version 0.4. The second edition ships with Wild Magic version 4.0, which when compared to version 0.1 looks very little like it. I believe the quality of the Wild Magic source code is a significant feature that has attracted many users. The latest version represents a significant rewrite to the rendering layer that has led to easier use of the engine and better performance by the renderers. The rewrite represents three months of dedicated time, something most authors would never consider investing time in, and it includes implementing a shader-based software renderer just to illustrate the book concepts in detail. I hope you enjoy using Wild Magic for your leisure projects!

1.3 A SUMMARY OF THE CHAPTERS

The book is partitioned into six parts.

Graphics. Chapter 2 discusses the details of a rendering system, including transformations, camera models, culling and clipping, rasterizing, and issues regarding software versus hardware rendering and about specific graphics application programmer interfaces (graphics APIs) in use these days. Chapter 3 is about rendering from the perspective of actually writing all the subsystems for a software renderer. The chapter includes what I consider a reasonable abstraction of the interface for a shader-based rendering system. This interface essentially shows that the renderer spends most of its time doing resource management. Chapter 3 also includes details about shader programs—not about writing them but about dealing with data management issues. Here I address such things as matching geometric vertex data to vertex program inputs, matching vertex program outputs to pixel program inputs, and ensuring that the infrastructure is set so that all resources are in the right place at the right time, hooked up, and ready to use for real-time rendering.

Scene Graph Management. Chapter 4 is about the essentials of organizing your data as a scene graph. This system is designed to be high level to allow ease of use

by application programmers, to be an efficient system to feed a renderer, and to be naturally extensible. The chapter also includes a section that talks about scene graph compiling—converting scene graphs to more optimized forms for target platforms. Chapters 5, 6, and 7 are about specially designed nodes and subsystems of the scene graph management system. These include subsystems to support animation, spatial sorting, and level of detail.

Collision Detection and Physics. Some general concepts you see in attempting to have physical realism in a three-dimensional (3D) application are discussed in Chapters 8 and 9. A generic approach to collision detection is presented, one that I have successfully implemented in a real environment. I also discuss picking operations and briefly talk about automatic pathfinding as a means for collision avoidance. The chapter on physics is a brief discussion of some concepts you will see often when working with physical simulations, but it does not include a discussion about the black-box-style physics you see in a commercial physics engine, such as Havok. That type of physics requires a lot more discussion than space allows in this book. Such physics is heavily mathematical and requires attention to issues of numerical round-off errors when using floating-point arithmetic.

Mathematical Topics. Chapters 10 through 17 include a lot of the mathematical detail for much of the source code you will find in Wild Magic. These chapters include a discussion of standard objects encountered in geometric manipulation and queries, including curves and surfaces covered in Chapters 11 and 12. You will also find material on queries regarding distance, containment, and intersection. Chapter 16 presents some common numerical methods that are useful in graphics and physics applications. The final chapter in this partition is about the topic of rotation, including basic properties of rotation matrices and how quaternions are related to matrices.

Software Engineering. Chapter 18 is a brief summary of basic principles of object-oriented design and programming. Various base-level support for large libraries is important and includes topics such as run-time type information, shared objects and reference counting, streaming of data (to/from disk, across a network), and initialization and termination for disciplined object creation and destruction in an application. Chapter 19 is about memory management. This is of particular importance when you want to write your own memory managers in order to build a system that is handed a fixed-size memory block and told it may only use memory from that block. In particular, this approach is used on game consoles where each engine (graphics, physics, sound, and so on) is given its memory “budget.” This is important for having predictable behavior of the engines. The last thing you want to happen in your game is one system consuming so much memory from a global heap that another system fails because it cannot successfully allocate what it needs.

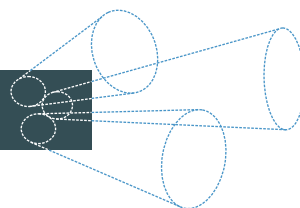
Special Effects Using Shaders. Chapter 20 shows a handful of sample shaders and the applications that use them. This is just to give you an idea of what you can do with shaders and how Wild Magic handles them. The appendix describes how you can add new shader effects to Wild Magic. The process is not difficult (by design).

I believe the organization here is an improvement over that of the first edition of the book. A number of valid criticisms of the first edition were about the amount

of mathematics interleaved in the discussions. Sorry, but I remain a firm believer that you need a lot of mathematics when building sophisticated graphics and physics engines. That said, I have made an attempt to discuss first the general concepts for graphics, factoring the finer detail into the chapters in the mathematics section that occurs late in the book. For example, it is possible to talk about culling of bounding volumes against frustum planes without immediately providing the details of the algorithm for specific bounding volumes. When discussing distance-based collision detection, it is possible to motivate the concepts without the specific distance algorithms for pairs of objects. The mathematics is still here, but factored to the end of the book rather than interleaved through the entire book.

Another criticism of the first edition of the book was its lack of figures. I believe I have remedied this, adding quite a few more figures to the second edition. That said, there may be places in the book where someone might feel the need for a figure where I thought the concept did not require one. My apologies if this happens. Send me feedback by email if you believe certain parts of the book can be improved by the addition of figures.

Finally, I have included some exercises in the book. Creating a large set of well-crafted exercises is a full-time job. In fact, I recall meeting a person who worked for Addison-Wesley (back in the early 1980s). His full-time job was managing the exercises for the calculus textbook by George Thomas and Ross Finney (seventh edition at that time). As much as I would like to have included more exercises here, my time budget for writing the book, writing the Wild Magic source code to go with the book, and making a living doing contract programming already exceeded 24 hours per day. I hope the exercises I have included will support the use of the book as a textbook in a graphics course. Most of them are programming exercises, requests to modify the source code to do something different or to do something in addition to what it does. I can imagine some of these taking quite some time to do. But I also believe they will make students think—the point of exercises!



THE GRAPHICS SYSTEM

This chapter provides some basic concepts that occur in a computer graphics system. Some of these concepts are mathematical in nature. I am assuming that you are familiar with trigonometry, vector and matrix algebra, and dot products and cross products. A warning to those who have a significant mathematical background: I intentionally discuss the mathematical concepts in a somewhat informal manner. My goal is to present the relevant ideas without getting tied down in the minutiae of stating rigorous definitions for the concepts. The first edition of this book was criticized for overemphasizing the mathematical details—and rightly so. Learn computer graphics first, and then later explore the beauty of formal mathematical exposition!

The foundations of coordinate systems (Section 2.1) and transformations (Section 2.2) are pervasive throughout a game engine. They are found not only in the graphics engines but in the physics engines and sound engines. Getting a model out of a modeling package and into the game world, setting up a camera for viewing, and displaying the model vertices and triangles is a process for which you must absolutely understand the coordinate systems and transformations. Scene graph management (Chapter 4) also requires a thorough understanding of these topics.

Sections 2.3 through 2.6 are the foundation for drawing 3D objects on a 2D screen. In a programming environment using graphics APIs such as OpenGL or Direct3D to access the graphics hardware, your participation in the process is typically restricted to selecting the parameters of the camera, providing the triangle primitives whose vertices have been assigned various attributes, and identifying objects that are not within the viewing region so that you do not have to draw them. The low-level processing of vertices and triangles is the responsibility of the graphics drivers. A discussion of the low-level processing is provided in this book, and a software renderer is part of the source code so you can see an actual implementation of the ideas.

Section 2.7 is a discussion about issues that are relevant when designing and implementing a graphics engine. The first edition of this book had a similar section, but I have added new material, further delineating how you must think when designing an engine or building a component to live on top of an existing graphics API. Section 2.7.2 is about the trade-offs you must consider if you want your code to be portable (or not). As you will see, the most important trade-off is which computational unit has the responsibility for certain operations.

Section 2.8 is about the vector and matrix conventions used by OpenGL, Direct3D, and Wild Magic. In your own code you must also choose conventions. These include how to store vectors and matrices, how they multiply together, how rotations apply, and so on. The section also mentions a few other conventions that make the APIs different enough that you need to pay attention to them when creating a cross-platform graphics engine.

2.1 THE FOUNDATION

We are all familiar with the notation of *tuples*. The standard 3-tuple is written as (x, y, z) . The components of the 3-tuple specify the location of a point in space relative to an origin. The components are referred to as the *Cartesian coordinates* of the point. You may have seen a diagram like the one in Figure 2.1 that illustrates the *standard coordinate system*.

Welcome to the book's first rendering of a 3D scene to a 2D screen, except this one was hand drawn! The standard coordinate system is simple enough, is it not? Coordinate systems other than the standard one may be imposed. Given that many people new to the field of computer graphics have some confusion about coordinate systems, perhaps it is not that simple after all. The confusion stems from having to work with multiple coordinate systems and knowing how they interact with each other. I will introduce these coordinate systems throughout the chapter and discuss their meaning. The important coordinate systems, called *spaces*, are *Cartesian space* (everything else is built on top of this), *model space* (or *object space*), *world space*, *view space* (or *camera space* or *eye space*), *clip space* (or *projection space* or *homogeneous space*), and *window space*.

Figure 2.1 is quite deceptive, which also leads to some confusion. I have drawn the figure as if the z -axis were in the upward direction. This is an unintentional consequence of having to draw *something* to illustrate a coordinate system. As humans who rely heavily on our vision, we have the notion of a *view direction*, an *up direction*, and a complementary *right direction* (sorry about that, left-handers). Just when you have become accustomed to thinking of the positive z -direction as the up direction, a modeling package comes along and insists that the positive y -direction is the up direction. The choice of view directions can be equally inconsistent, especially the defaults for various graphics engines and APIs. It is important to understand the co-

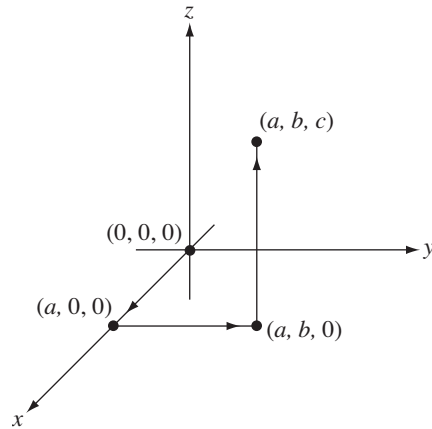


Figure 2.1 The standard coordinate system in three dimensions. The point at (a, b, c) is reached by starting at the origin $(0, 0, 0)$, moving a units in the direction $(1, 0, 0)$ to the point $(a, 0, 0)$, then moving b units in the direction $(0, 1, 0)$ to the point $(a, b, 0)$, and then moving c units in the direction $(0, 0, 1)$ to the point (a, b, c) .

ordinate system conventions for all the packages you use in your game development. Throughout the book, my discussions about coordinate systems will refer to view, up, and right directions with coordinate names d , u , and r , respectively, rather than to axis names such as x , y , and z .

2.1.1 COORDINATE SYSTEMS

Rather than constantly writing tuples, it is convenient to have a shorter notation to represent points and vectors. I will use boldface to do so. For example, the tuple (x, y, z) can refer to a point named $\mathbf{P}$. Although mathematicians distinguish between a *point* and the *coordinates of a point*, I will be loose with the notation here and simply say $\mathbf{P} = (x, y, z)$. The typical names I use for the direction vectors are $\mathbf{D}$ for the view direction, $\mathbf{U}$ for the up direction, and $\mathbf{R}$ for the right direction. Using the standard convention, a direction vector must have unit length. For example, $(1, 0, 0)$ is a direction vector, but $(1, 1, 1)$ is not since its length is $\sqrt{3}$. In our agreed-upon notation, we can now define a coordinate system and how points are represented within a coordinate system.

A *coordinate system* consists of an *origin* point $\mathbf{E}$ and three independent direction vectors, $\mathbf{D}$ (view direction), $\mathbf{U}$ (up direction), and $\mathbf{R}$ (right direction). You may think

of the origin as the location of an observer who wishes to make measurements from his own perspective. The coordinate system is written succinctly as

$$\{\mathbf{E}; \mathbf{D}, \mathbf{U}, \mathbf{R}\} \quad (2.1)$$

Any point $\mathbf{X}$ may be represented in the coordinate system as

$$\mathbf{X} = \mathbf{E} + d\mathbf{D} + u\mathbf{U} + r\mathbf{R} \quad (2.2)$$

where d , u , and r are scalars that measure how far along the related direction you must move to get to the point $\mathbf{X}$. The tuple (d, u, r) lists the coordinates of $\mathbf{X}$ relative to the coordinate system in Equation (2.1).

The direction vectors are nearly always chosen to be mutually perpendicular. This is not necessary for coordinate systems. All that matters is that the directions are independent (*linearly independent* to those of you with some training in linear algebra). In this book, I will assume that the directions are indeed mutually perpendicular. If for any reason I need a coordinate system that does not have this property, I will make it very clear to you in that discussion. The assumption that the direction vectors in the coordinate system of Equation (2.1) are unit length and mutually perpendicular allows us to easily solve for the coordinates

$$d = \mathbf{D} \cdot (\mathbf{X} - \mathbf{E}), \quad u = \mathbf{U} \cdot (\mathbf{X} - \mathbf{E}), \quad r = \mathbf{R} \cdot (\mathbf{X} - \mathbf{E}) \quad (2.3)$$

where the bullet symbol $(\cdot)$ denotes the dot product of vectors. The construction of the coefficients relies on the directions having unit length ($\mathbf{D} \cdot \mathbf{D} = \mathbf{U} \cdot \mathbf{U} = \mathbf{R} \cdot \mathbf{R} = 1$) and being mutually perpendicular ($\mathbf{D} \cdot \mathbf{U} = \mathbf{D} \cdot \mathbf{R} = \mathbf{U} \cdot \mathbf{R} = 0$). In addition to being mutually perpendicular, the direction vectors are assumed to form a *right-handed system*. Specifically, I require that $\mathbf{R} = \mathbf{D} \times \mathbf{U}$, where the times symbol $(\times)$ denotes the cross product operator. This condition quantifies the usual *right-hand rule* for computing a cross product. A coordinate system may also be constructed to be a *left-handed system* using the *left-hand rule* for computing a cross product.

But wait. What does it really mean to be right-handed or left-handed? And what really is a cross product? The concepts have both algebraic and geometric interpretations, and I have seen many times where the two interpretations are misunderstood. This is the topic I want to analyze next.

2.1.2 HANDEDNESS AND CROSS PRODUCTS

To muddy the waters of coordinate system terminology, the Direct3D documentation [Cor] has a section

DirectX Graphics | Direct3D 9 | Programming Guide | Getting Started |
Coordinate Systems and Geometry | Coordinate Systems

This section is a one-page description of coordinate systems, but unfortunately it is sparse on details and is not precise in its language. The documentation describes right-handed versus left-handed coordinate systems, but assumes that the positive y -axis is the up direction, the positive x -axis is the right direction, and the positive z -axis points into the plane of the page for left-handed coordinates but out of the plane of the page for right-handed coordinates. Later in the documentation you will find this quote:

Although left-handed and right-handed coordinates are the most common systems, there is a variety of other coordinate systems used in 3D software.

Coordinate systems are either left-handed or right-handed: there are no other choices. The remainder of the quote is

For example, it is not unusual for 3D modeling applications to use a coordinate system in which the y -axis points toward or away from the viewer, and the z -axis points up. In this case, right-handedness is defined as any positive axis (x , y , or z) pointing toward the viewer. Left-handedness is defined as any positive axis (x , y , or z) pointing away from the viewer.

The terminology here is imprecise. First, coordinate systems may be chosen for which none of the axis direction vectors are the x -, y -, or z -axes. Second, handedness has to do with the order in which you list your vectors and components. The way you draw your coordinate system in a figure is intended to illustrate the handedness, not to define the handedness.

Let us attempt to make the notions precise. The underlying structure for everything we do in three dimensions is the *tuple*. Essentially, we all assume the existence of Cartesian space, as described previously and illustrated in Figure 2.1. Cartesian space is the one on which *all of us* base our coordinate systems. The various spaces such as model space, world space, and view space are all built on top of Cartesian space by specifying a coordinate system. More importantly, *Cartesian space has no preferential directions for view, up, or right*. Those directions are what you specify *when you impose a coordinate system on Cartesian space for an observer to use*.

I have already presented an intuitive way to specify a coordinate system in Equation (2.1). I have imposed the requirement that the coordinate system is right-handed. But what does this really mean? The *geometric interpretation* is shown in Figure 2.2 (a).

In Figure 2.2 (a), the placement of $\mathbf{R}$ relative to $\mathbf{D}$ and $\mathbf{U}$ follows the right-hand rule. The *algebraic interpretation* is the equation $\mathbf{R} = \mathbf{D} \times \mathbf{U}$, which uses the following definition. Given two Cartesian tuples, (x_0, y_0, z_0) and (x_1, y_1, z_1) , the cross product is defined by

$$(x_0, y_0, z_0) \times (x_1, y_1, z_1) = (y_0z_1 - z_0y_1, z_0x_1 - x_0z_1, x_0y_1 - y_0x_1) \quad (2.4)$$

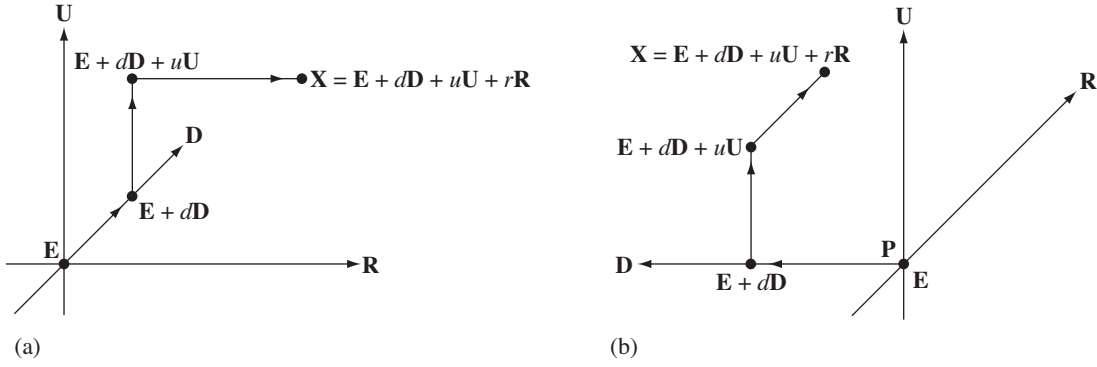


Figure 2.2 (a) A geometric illustration of the right-handed coordinate system in Equation (2.1).
 (b) A geometric illustration of the left-handed coordinate system in Equation (2.5).

Both the algebraic interpretation and the geometric interpretation are founded on the standard Cartesian coordinate system.

A point X is represented in the coordinate system of Equation (2.1) via Equation (2.2), where the components of the coordinate tuple (d, u, r) are computed by Equation (2.3). Instead, I could have used the coordinate system

$$\{E; R, U, D\} \quad (2.5)$$

with the representation of X given by

$$X = E + rR + uU + dD \quad (2.6)$$

This coordinate system is left-handed and the coordinate tuple for X is (r, u, d) . Algebraically, Equations (2.2) and (2.6) produce the same point X . All that is different is the bookkeeping, so to speak, which manifests itself as a geometric property as illustrated by Figure 2.2. In the right-handed system shown in Figure 2.2 (a), D points into the plane of the page and the last coordinate direction R points to the right. It is the case that $D \times U = R$; the last vector is the cross product of the first two vectors. In the left-handed system shown in Figure 2.2 (b), R points into the plane of the page and the last coordinate direction D points to the left. It is the case that $R \times U = -D$; the last vector is the negative of the cross product of the first two vectors.

Whereas Figure 2.2 illustrates the geometric difference between left-handed and right-handed coordinate systems, the algebraic way to classify whether a coordinate system is left-handed or right-handed is via the cross product operation. Generally, if you have a coordinate system

$$\{P; U_0, U_1, U_2\}$$

where $\mathbf{P}$ is the origin and where the $\mathbf{U}_i$ are unit length and mutually perpendicular, then the coordinate system is *right-handed* when

$$\mathbf{U}_0 \times \mathbf{U}_1 = \mathbf{U}_2 \quad (\text{right-handed})$$

and is *left-handed* when

$$\mathbf{U}_0 \times \mathbf{U}_1 = -\mathbf{U}_2 \quad (\text{left-handed})$$

Equivalently, if you write the coordinate direction vectors as the columns of a matrix, say, $Q = [\mathbf{U}_0 \ \mathbf{U}_1 \ \mathbf{U}_2]$, then Q is an orthogonal matrix. Right-handed systems occur when $\det(Q) = 1$ and left-handed systems occur when $\det(Q) = -1$.

I have more to say about this discussion. Sometimes I read news posts where people say that Direct3D has a “left-handed cross product.” This is imprecise terminology, and I will explain why, using an example. Let $\mathbf{A}$ and $\mathbf{B}$ be vectors (not points). The representations of the vectors relative to the coordinate system of Equation (2.1) are

$$\mathbf{A} = d_a \mathbf{D} + u_a \mathbf{U} + r_a \mathbf{R}, \quad \mathbf{B} = d_b \mathbf{D} + u_b \mathbf{U} + r_b \mathbf{R}$$

and their cross product is

$$\mathbf{A} \times \mathbf{B} = (u_a r_b - r_a u_b) \mathbf{D} + (r_a d_b - d_a r_b) \mathbf{U} + (d_a u_b - u_a d_b) \mathbf{R} \quad (2.7)$$

The representations of the vectors relative to the coordinate system of Equation (2.5) are

$$\mathbf{A} = r_a \mathbf{R} + u_a \mathbf{U} + d_a \mathbf{D}, \quad \mathbf{B} = r_b \mathbf{R} + u_b \mathbf{U} + d_b \mathbf{D}$$

and their cross product is

$$\mathbf{A} \times \mathbf{B} = (d_a u_b - u_a d_b) \mathbf{R} + (r_a d_b - d_a r_b) \mathbf{U} + (u_a r_b - r_a u_b) \mathbf{D} \quad (2.8)$$

Regardless of which coordinate system was used, Equations (2.7) and (2.8) produce the same vector in Cartesian space.

Now let’s work with the coordinates themselves, but with specific instances just to simplify the discussion. Let $\mathbf{A} = \mathbf{D}$ and $\mathbf{B} = \mathbf{U}$. In the coordinate system of Equation (2.1), the coordinate tuple of $\mathbf{A}$ is (1, 0, 0). This says that you have 1 of $\mathbf{D}$ and none of the other two vectors. The coordinate tuple of $\mathbf{B}$ is (0, 1, 0). This says you have 1 of $\mathbf{U}$ and none of the other two vectors. According to Equation (2.4), the cross product of these tuples is

$$(1, 0, 0) \times (0, 1, 0) = (0, 0, 1) \quad (2.9)$$

In the coordinate system of Equation (2.5), the coordinate tuple of $\mathbf{A}$ is (0, 0, 1) since $\mathbf{D}$ is the last vector in the list of coordinate axis directions. The coordinate tuple of $\mathbf{B}$

is $(0, 1, 0)$. According to Equation (2.4), the cross product of these tuples is

$$(0, 0, 1) \times (0, 1, 0) = (-1, 0, 0) \quad (2.10)$$

Whereas Equations (2.7) and (2.8) produce the same Cartesian tuple, Equations (2.9) and (2.10) produce different tuples. Is this a contradiction? No. The tuples $(0, 0, 1)$ and $(-1, 0, 0)$ are not for the Cartesian space; they are coordinate tuples relative to the coordinate systems imposed on Cartesian space. The tuple $(0, 0, 1)$ is relative to the right-handed coordinate system of Equation (2.1), so the actual Cartesian tuple is $0\mathbf{D} + 0\mathbf{U} + 1\mathbf{R} = \mathbf{R}$; that is, $\mathbf{A} \times \mathbf{B} = \mathbf{D} \times \mathbf{U} = \mathbf{R}$. In Figure 2.2 (a), you obtain $\mathbf{R}$ from $\mathbf{D}$ and $\mathbf{U}$ by using the right-hand rule. Similarly, the tuple $(-1, 0, 0)$ is relative to the left-handed coordinate system of Equation (2.5), so the actual Cartesian tuple is $-1\mathbf{R} + 0\mathbf{U} + 0\mathbf{D} = -\mathbf{R}$; that is, $\mathbf{A} \times \mathbf{B} = \mathbf{D} \times \mathbf{U} = -\mathbf{R}$. In Figure 2.2 (b), you obtain $-\mathbf{R}$ from $\mathbf{D}$ and $\mathbf{U}$ by using the left-hand rule. Table 2.1 summarizes our findings when $\mathbf{A} = \mathbf{D}$ and $\mathbf{B} = \mathbf{U}$.

If you compute cross products using coordinate tuples, as shown in Equation (2.10) for a left-handed camera coordinate system, you have a left-handed cross product, so to speak. But if you compute cross products using right-handed Cartesian tuples, as shown in Equation (2.8) also for a left-handed camera coordinate system, you wind up with a right-handed cross product, so to speak. This means you have to be very careful when computing cross products, making certain you know *which* coordinate system you are working with. The `Direct3D` function for computing the cross product does not care about the coordinate system:

```
D3DXVECTOR A = <a tuple (x0,y0,z0)>;
D3DXVECTOR B = <a tuple (x1,y1,z1)>;
D3DXVECTOR C;
D3DXVec3Cross(&C,&A,&B); // C = (y0 z1 - z0 y1, z0 x1 - x0 z1, x0 y1 - y0 x1)
```

The function simply implements Equation (2.4) without regard to which coordinate system the “tuples” $\mathbf{A}$ and $\mathbf{B}$ come from. The function knows algebra. You are the one who imposes geometry.

Table 2.1 Summary of handedness and cross product calculations.

<i>Equation</i>	<i>Result</i>	<i>Coordinate System Handedness</i>	<i>Cross Product Applied To</i>
(2.7)	$\mathbf{D} \times \mathbf{U} = \mathbf{R}$	Right-handed	Cartesian tuples
(2.8)	$\mathbf{D} \times \mathbf{U} = \mathbf{R}$	Left-handed	Cartesian tuples
(2.9)	$\mathbf{D} \times \mathbf{U} = \mathbf{R}$	Right-handed	Coordinate tuples
(2.10)	$\mathbf{D} \times \mathbf{U} = -\mathbf{R}$	Left-handed	Coordinate tuples

2.1.3 POINTS AND VECTORS

You might have noticed that I have referred to points and to vectors. These two concepts are considered distinct and are the topics of *affine algebra*. A point is *not* a vector and a vector is *not* a point. To distinguish between points and vectors within text, some authors use different fonts. For the sake of argument, let us do so and consider three points $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$ and a vector $\mathbf{V}$. The following operations are axioms associated with affine algebra, but once again to be loose with the notation, I will just say the following:

1. The difference of the two points is a vector, $\mathbf{V} = \mathcal{P} - \mathcal{Q}$.
2. The sum of a point and a vector is a point, $\mathcal{Q} = \mathcal{P} + \mathbf{V}$.
3. $(\mathcal{P} - \mathcal{Q}) + (\mathcal{Q} - \mathcal{R}) = (\mathcal{P} - \mathcal{R})$. The intuition for this is to draw a triangle whose vertices are the three points. This equation says that the sum of the directed edges of the triangle is the zero vector.

In the third axiom, rather than appealing to the geometry of a triangle, you might be tempted to remove the parentheses. In fact, the removal of the parentheses is a *consequence* of this axiom, but you have to be careful in doing such things that, at first glance, seem intuitive. The axioms do not allow you to *add points* in any manner you like; for example, the expression $\mathcal{P} + \mathcal{Q}$ is not valid. However, additional axioms may be postulated that allow a special form of addition in expressions called *affine combinations*. Specifically, the following axioms support this:

4. $\mathcal{P} = \sum_{i=1}^n c_i \mathcal{P}_i$ is a point, where $\sum_{i=1}^n c_i = 1$.
5. $\mathbf{V} = \sum_{i=1}^n d_i \mathcal{P}_i$ is a vector, where $\sum_{i=1}^n d_i = 0$.

With the additional axioms, an expression such as

$$(1/3)\mathcal{P} + (1/3)\mathcal{Q} + (1/3)\mathcal{R}$$

is valid since the coefficients sum to one. This example produces the average of the points. The expression

$$(\mathcal{P} - \mathcal{Q}) + (\mathcal{Q} - \mathcal{R}) = \mathcal{P} - \mathcal{Q} + \mathcal{Q} - \mathcal{R}$$

is valid. The implied coefficients of the points on the right-hand side are 1, -1 , 1, and -1 (in that order), and their sum is zero. Thus, the expression on the right-hand side is a vector.

At the beginning of this chapter, I promised not to delve into the finer mathematical details of the topics. I have done so here, but for practical reasons. Some graphics programmers choose to enforce the distinction between points and vectors, say, in an object-oriented language such as C++. To hint at the topic of Section 2.2.5, a fourth component is provided for both points and vectors. Points are represented as

4-tuples of the form $(x, y, z, 1)$ and vectors are represented as 4-tuples of the form $(x, y, z, 0)$. The five axioms for affine algebra are satisfied by these representations.

Axiom 1 is satisfied,

$$(x_0, y_0, z_0, 1) - (x_1, y_1, z_1, 1) = (x_0 - x_1, y_0 - y_1, z_0 - z_1, 0) \quad (2.11)$$

Axiom 2 is satisfied,

$$(x_0, y_0, z_0, 1) + (x_1, y_1, z_1, 0) = (x_0 + x_1, y_0 + y_1, z_0 + z_1, 1) \quad (2.12)$$

Axiom 3 is satisfied,

$$\begin{aligned} & ((x_0, y_0, z_0, 1) - (x_1, y_1, z_1, 1)) + ((x_1, y_1, z_1, 1) - (x_2, y_2, z_2, 1)) \\ &= (x_0 - x_1, y_0 - y_1, z_0 - z_1, 0) + (x_1 - x_2, y_1 - y_2, z_1 - z_2, 0) \\ &= (x_0 - x_2, y_0 - y_2, z_0 - z_2, 0) \\ &= ((x_0, y_0, z_0, 1) - (x_2, y_2, z_2, 1)) \end{aligned}$$

Axiom 4 is satisfied,

$$\begin{aligned} \sum_{i=1}^n c_i (x_i, y_i, z_i, 1) &= \left(\sum_{i=1}^n c_i x_i, \sum_{i=1}^n c_i y_i, \sum_{i=1}^n c_i z_i, \sum_{i=1}^n c_i \right) \\ &= \left(\sum_{i=1}^n c_i x_i, \sum_{i=1}^n c_i y_i, \sum_{i=1}^n c_i z_i, 1 \right) \end{aligned} \quad (2.13)$$

where I used the fact that the c_i sum to one. Axiom 5 is satisfied,

$$\begin{aligned} \sum_{i=1}^n d_i (x_i, y_i, z_i, 1) &= \left(\sum_{i=1}^n d_i x_i, \sum_{i=1}^n d_i y_i, \sum_{i=1}^n d_i z_i, \sum_{i=1}^n d_i \right) \\ &= \left(\sum_{i=1}^n d_i x_i, \sum_{i=1}^n d_i y_i, \sum_{i=1}^n d_i z_i, 0 \right) \end{aligned} \quad (2.14)$$

where I used the fact that the d_i sum to zero.

Two classes are implemented, one called `Point` and one called `Vector`. The interface for the `Vector` class has minimally the following structure:

```
class Vector
{
public:
    // 'this' is vector U
    Vector operator+ (Vector V) const;           // U + V
    Vector operator- (Vector V) const;           // U - V
```

```

    Vector operator* (float c) const;           // V*c
    friend Vector operator* (float c, Vector V) const; // c*V

private:
    float tuple[4]; // tuple[3] = 0 always
};

```

The interface for the Point class has minimally the following structure:

```

class Point
{
public:
    // 'this' is point P
    Point operator+ (Vector V); // P + V, Equation (2.12)
    Point operator- (Vector V); // P - V, Equation (2.12)
    Vector operator- (Point Q); // P - Q, Equation (2.11)
    static Point AffineCSum (int N, float c[], Point Q[]); // Equation (2.13)
    static Vector AffineDSum (int N, float d[], Point Q[]); // Equation (2.14)

private:
    float tuple[4]; // tuple[3] = 1 always
};

```

I have shown the points and vectors stored as 4-tuples. The fourth component should be private so that applications cannot access them and inadvertently change them. Thus, if you were to support an `operator[]` member function, you would need to trap attempts to access the fourth component (via assertions, exceptions, or some other mechanism). It is possible to use 3-tuples, relying on the fact that the class names themselves imply the correct fourth component. Given current CPUs and game consoles, it is better, though, to have 16-byte (4-float) alignment of data because the hardware expects it in order to perform well.

Two issues come to mind. First, having classes `Point` and `Vector` means maintaining more code than having just a single class to represent points and vectors. The amount of additional source code might not be of concern to you. Second, and perhaps the more important issue, is that the clipping process occurs using 4-tuples of the form (x, y, z, w) , where the fourth component w is not necessarily 0 or 1. Clipping involves arithmetic operations on 4-tuples, but the `Point` class does not support this when it insists on $w = 1$. If you were to allow the fourth component to be public and not force it to be 1, you would be violating the purist attempt to distinguish between points and vectors. You could implement yet another class, say, `HPoint`, and represent the general 4-tuples, but this means maintaining even more code. My opinion is to keep it simple and just support a vector class, keeping the distinction between points and vectors in your own mind, being consistent about it when coding, and not maintaining additional code.

2.2 TRANSFORMATIONS

We want to construct functions that map points in 3D space to other points in 3D space. The motivation was provided in the last section: taking an object built in model space and placing it in world space. The transformations considered in this section are the simplest ones you encounter in computer graphics.

2.2.1 LINEAR TRANSFORMATIONS

The topic of *linear transformations* is usually covered in a course on linear algebra. Such transformations are applied to vectors rather than points. I will not give a detailed overview here. You can read about the topic in any standard textbook on linear algebra.

The basic idea is to construct functions of the form $\mathbf{Y} = \mathbf{L}(\mathbf{X})$. The input to the function is the vector $\mathbf{X}$ and the output is the vector $\mathbf{Y}$. The function name itself is also typeset as a vector, namely, $\mathbf{L}$, to indicate that its output is a vector. A *linear function* or *linear transformation* is defined to have the following property:

$$\mathbf{L}(c\mathbf{U} + \mathbf{V}) = c\mathbf{L}(\mathbf{U}) + \mathbf{L}(\mathbf{V}) \quad (2.15)$$

where c is a scalar. The expression $c\mathbf{U} + \mathbf{V}$ is called a *linear combination* of the two vectors. In words, Equation (2.15) says that the function value of a linear combination is the linear combination of the function values.

EXAMPLE
2.1

Let $\mathbf{X} = (x_0, x_1, x_2)$ and $\mathbf{Y} = (y_0, y_1, y_2)$. The function $\mathbf{Y} = \mathbf{L}(\mathbf{X})$, where $\mathbf{L}(x_0, x_1, x_2) = (x_0 + x_1, 2x_0 - x_2, 3x_0 + x_1 + 2x_2)$, is a linear transformation. To verify this, apply the function to the linear combination $c\mathbf{U} + \mathbf{V}$, where $\mathbf{U} = (u_0, u_1, u_2)$, $\mathbf{V} = (v_0, v_1, v_2)$, and c is a scalar:

$$\begin{aligned} \mathbf{L}(c\mathbf{U} + \mathbf{V}) &= \mathbf{L}(c(u_0, u_1, u_2) + (v_0, v_1, v_2)) \\ &= \mathbf{L}(cu_0 + v_0, cu_1 + v_1, cu_2 + v_2) \\ &= ((cu_0 + v_0) + (cu_1 + v_1), 2(cu_0 + v_0) - (cu_2 + v_2), \\ &\quad 3(cu_0 + v_0) + (cu_1 + v_1) + 2(cu_2 + v_2)) \\ &= (c(u_0 + u_1) + (v_0 + v_1), c(2u_0 - u_2) + (2v_0 - v_2), \\ &\quad c(3u_0 + u_1 + 2u_2) + (3v_0 + v_1 + 2v_2)) \\ &= c(u_0 + u_1, 2u_0 - u_2, 3u_0 + u_1 + 2u_2) \\ &\quad + (v_0 + v_1, 2v_0 - v_2, 3v_0 + v_1 + 2v_2) \\ &= c\mathbf{L}(u_0, u_1, u_2) + \mathbf{L}(v_0, v_1, v_2) \\ &= c\mathbf{L}(\mathbf{U}) + \mathbf{L}(\mathbf{V}) \end{aligned}$$

These algebraic steps verify that $L(c\mathbf{U} + \mathbf{V}) = cL(\mathbf{U}) + L(\mathbf{V})$, so the function is linear. ■

EXAMPLE
2.2

The function $L(x_0, x_1, x_2) = x_0^2$ is not a linear transformation. To verify this, it is enough to show that the function applied to *one* linear combination is not the linear combination of the function results. For example, let $c = 2$, $\mathbf{U} = (1, 0, 0)$, and $\mathbf{V} = (0, 0, 0)$; then $L(\mathbf{U}) = L(1, 0, 0) = 1$, $L(\mathbf{V}) = L(0, 0, 0) = 0$, and $L(c\mathbf{U} + \mathbf{V}) = L(2, 0, 0) = 4$. Also, $2L(\mathbf{U}) + L(\mathbf{V}) = 2L(1, 0, 0) + L(0, 0, 0) = 2$. This specific case does not satisfy the constraint $L(c\mathbf{U} + \mathbf{V}) = cL(\mathbf{U}) + L(\mathbf{V})$, so the function is not linear. ■

EXAMPLE
2.3

Translation of a vector is not linear. The translation function is $L(x_0, x_1, x_2) = (x_0, x_1, x_2) + (b_0, b_1, b_2)$, where $(b_0, b_1, b_2) \neq (0, 0, 0)$ is the vector used to translate any point in space. If you choose $c = 1$, $\mathbf{U} = (1, 1, 1)$, and $\mathbf{V} = (0, 0, 0)$, then

$$L(\mathbf{U} + \mathbf{V}) = L(1, 1, 1) = (1 + b_0, 1 + b_1, 1 + b_2)$$

which is different from

$$\begin{aligned} L(\mathbf{U}) + L(\mathbf{V}) &= L(1, 1, 1) + L(0, 0, 0) = (1 + b_0, 1 + b_1, 1 + b_2) + (b_0, b_1, b_2) \\ &= (1 + 2b_0, 1 + 2b_1, 1 + 2b_2) \end{aligned}$$

Translation is, however, an example of an *affine transformation*, which I discuss later in this section. ■

Linear transformations are convenient to use because they have a representation that makes them easy to implement and compute in a program. Let us write the transformation input $\mathbf{X}$ and output $\mathbf{Y}$ as column vectors (3×1 vectors). A linear transformation is necessarily of the form

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} m_{00}x_0 + m_{01}x_1 + m_{02}x_2 \\ m_{10}x_0 + m_{11}x_1 + m_{12}x_2 \\ m_{20}x_0 + m_{21}x_1 + m_{22}x_2 \end{bmatrix} \quad (2.16)$$

where the coefficients m_{ij} of the 3×3 matrix are constants. The more compact notation is

$$\mathbf{Y} = M\mathbf{X} \quad (2.17)$$

where M is a 3×3 matrix. This form is suggestive of the one-dimensional case $y = mx$, which is the equation of a straight line that passes through the origin. The geometry of linear functions in higher dimensions is slightly more complicated. Once again, I refer you to any standard textbook on linear algebra to see the details.

By using the representation $\mathbf{Y} = M\mathbf{X}$, I have already chosen a convention regarding the manipulation of vectors and matrices. Vectors are columns for me, and the application of a matrix to a vector puts the matrix on the left and the vector on the

right. This is the mathematician in me speaking—I chose what I was raised with. OpenGL and Direct3D choose the opposite convention, which is to represent vectors as rows and to apply a matrix to a vector by putting the vector on the left and the matrix on the right. This is the usual convention chosen by computer graphics people. There is no right or wrong for choosing your conventions. For any choice you make, you will find users who disagree with that choice because they have made another choice. The fact is, *you* make your decisions. *You* live by the consequences. What is important is to ensure that you have documented your engine and code well, making it clear to clients exactly what your choices are. There are quite a few other conventions you must decide on when designing a computer graphics system. A comparison of the conventions for transformations is provided in Section 2.8 regarding Wild Magic, OpenGL, and Direct3D.

EXERCISE
2.1

Verify that the function defined by Equation (2.16) is a linear transformation. ■

Computer graphics has a collection of linear transformations that arise frequently in practice. These are presented here.

Rotation

The motivation for 3D rotation comes from two dimensions, where a rotation matrix is

$$R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} = I + (\sin \theta)S + (1 - \cos \theta)S^2 \quad (2.18)$$

where I is the identity matrix and S is the skew-symmetric matrix, as shown:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad S = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

I have chosen to factor the matrix R in terms of I , S , and θ because it is suggestive of how to build the matrix for rotation about an arbitrary axis. For a positive angle θ , RV rotates the 2×1 vector V *counterclockwise* about the origin, as shown in Figure 2.3. Whether a positive angle represents a counterclockwise or a clockwise rotation is yet another convention you must choose and make clear to your users.

In three dimensions, the matrix representing a rotation in the xy -plane is

$$R = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = I + (\sin \theta)S + (1 - \cos \theta)S^2 \quad (2.19)$$

where

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad S = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Assuming a choice of coordinate axes as is shown in Figure 2.1, the direction of rotation is counterclockwise about the z -axis when viewed by an observer who is on the positive z -side of the xy -plane and looking at the plane with view direction in the negative z -direction, $(0, 0, -1)$. Think of Figure 2.3 as what such an observer sees. In that figure, the positive z -direction is out of the page; the negative z -direction is into the page. A 3D view is shown in Figure 2.4 (a).

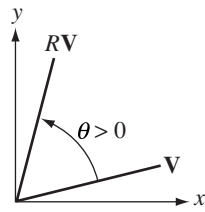


Figure 2.3 A positive angle corresponds to a counterclockwise rotation.

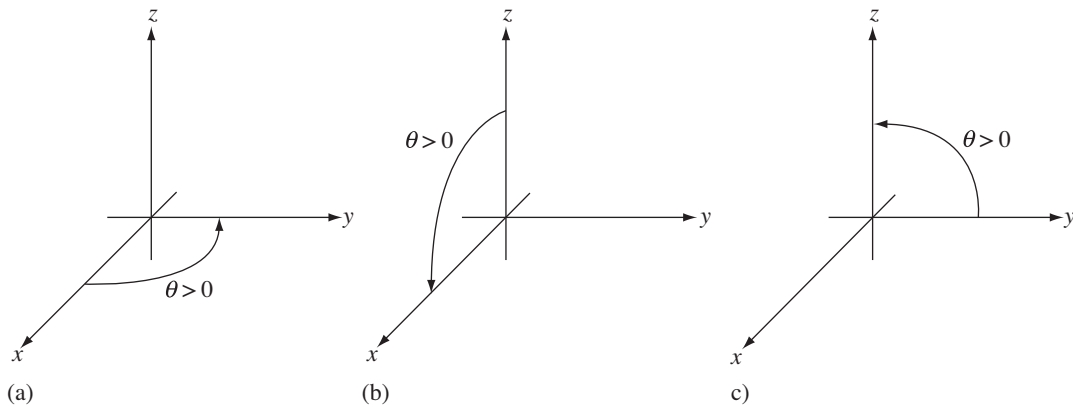


Figure 2.4 Rotations about the coordinate axes. (a) A positive-angle rotation about the z -axis. (b) A positive-angle rotation about the y -axis. (c) A positive-angle rotation about the x -axis.

Similar rotation matrices may be constructed for rotations about the other coordinate axes. The matrix representing a rotation in the xz -plane is

$$R = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix} = I + (\sin \theta)S + (1 - \cos \theta)S^2 \quad (2.20)$$

where

$$S = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

Figure 2.4 (b) is a 3D view of a positive-angle rotation about the y -axis, which is a counterclockwise rotation in the xz -plane as shown. The matrix representing a rotation in the yz -plane is

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix} = I + (\sin \theta)S + (1 - \cos \theta)S^2 \quad (2.21)$$

where

$$S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

Figure 2.4 (c) is a 3D view of a positive-angle rotation about the x -axis, which is a counterclockwise rotation in the yz -plane as shown.

In general, the matrix representing a rotation about the axis with direction vector (u_0, u_1, u_2) is as shown. Define the skew-symmetric matrix

$$S = \begin{bmatrix} 0 & -u_2 & u_1 \\ u_2 & 0 & -u_0 \\ -u_1 & u_0 & 0 \end{bmatrix}$$

then the rotation matrix is

$$\begin{aligned} R &= I + (\sin \theta)S + (1 - \cos \theta)S^2 \\ &= \begin{bmatrix} \gamma + (1 - \gamma)u_0^2 & -u_2\sigma + (1 - \gamma)u_0u_1 & +u_1\sigma + (1 - \gamma)u_0u_2 \\ +u_2\sigma + (1 - \gamma)u_0u_1 & \gamma + (1 - \gamma)u_1^2 & -u_0\sigma + (1 - \gamma)u_1u_2 \\ -u_1\sigma + (1 - \gamma)u_0u_2 & +u_0\sigma + (1 - \gamma)u_1u_2 & \gamma + (1 - \gamma)u_2^2 \end{bmatrix} \end{aligned} \quad (2.22)$$

where $\sigma = \sin \theta$ and $\gamma = \cos \theta$.

EXERCISE 2.2 Verify that Equation (2.22) produces the coordinate plane rotations mentioned in Equations (2.19), (2.20), and (2.21). Also verify that the S -matrix in Equation (2.22) produces the S -matrices for the coordinate plane rotations. ■

EXERCISE 2.3 Using algebraic methods, construct the formula of Equation (2.22). ■

Reflection

A plane passing through the origin is represented algebraically by the equation $\mathbf{N} \cdot \mathbf{X} = 0$, where $\mathbf{N}$ is a unit-length vector perpendicular to the plane and $\mathbf{X}$ is any point on the plane. Figure 2.5 (a) provides a 3D view of the plane, a vector $\mathbf{V}$, and the reflection $\mathbf{U}$ of $\mathbf{V}$ through the plane. Figure 2.5 (b) shows a 2D side view.

The vector $\mathbf{N}^\perp$ is a point on the plane and is the midpoint of the line segment connecting $\mathbf{V}$ and its reflection $\mathbf{U}$. The superscript symbol $\perp$ denotes perpendicularity. In this case, $\mathbf{N}^\perp$ is a vector perpendicular to $\mathbf{N}$.

The side view gives you a good idea of how the vectors are related algebraically. The input vector is the linear combination

$$\mathbf{V} = c\mathbf{N} + \mathbf{N}^\perp$$

for the scalar c . The perpendicular component $\mathbf{N}^\perp$ is naturally dependent on your choice of $\mathbf{V}$. In fact, the scalar is easily determined to be $c = \mathbf{N} \cdot \mathbf{V}$, which uses the conditions that $\mathbf{N}$ is unit length and that $\mathbf{N}$ and $\mathbf{N}^\perp$ are perpendicular. The reflection vector is the linear combination

$$\mathbf{U} = -c\mathbf{N} + \mathbf{N}^\perp$$

The difference is that the normal component of $\mathbf{V}$ is negated to form the normal component for $\mathbf{U}$; this is the reflection. Subtracting the two equations and using the

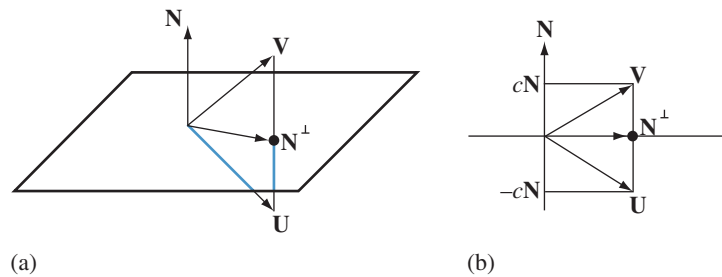


Figure 2.5 The reflection of a vector through a plane. (a) A 3D view. (b) A 2D side view.

formula for c leads to

$$\mathbf{U} = \mathbf{V} - 2(\mathbf{N} \cdot \mathbf{V})\mathbf{N} = (\mathbf{I} - 2\mathbf{N}\mathbf{N}^T) \mathbf{V}$$

where $\mathbf{I}$ is the 3×3 identity matrix and the superscript T denotes the transpose operation. Using my conventions, $\mathbf{N}$ is a 3×1 (column) vector, which makes $\mathbf{N}^T$ a 1×3 (row) vector. The product $\mathbf{N}\mathbf{N}^T$ is a 3×3 matrix. Notice that $\mathbf{N}^T\mathbf{N}$ is the product in the other order but is necessarily a scalar (a 1×1 matrix as it were).

If $\mathbf{N} = (n_0, n_1, n_2)$, the reflection matrix is

$$R = \mathbf{I} - \mathbf{N}\mathbf{N}^T = \begin{bmatrix} 1 - n_0^2 & -n_0n_1 & -n_0n_2 \\ -n_0n_1 & 1 - n_1^2 & -n_1n_2 \\ -n_0n_2 & -n_1n_2 & 1 - n_2^2 \end{bmatrix} \quad (2.23)$$

Rotation and reflection matrices are said to be *orthogonal matrices*. An orthogonal matrix M has the property that $MM^T = M^TM = \mathbf{I}$, which says that the inverse operation of M is its transpose. Apply M to a vector $\mathbf{V}$ to obtain $\mathbf{U} = M\mathbf{V}$. Now apply M^T to obtain $M^T\mathbf{U} = M^TM\mathbf{V} = \mathbf{I}\mathbf{V} = \mathbf{V}$. Geometrically, if R is a rotation matrix that rotates a vector about an axis by θ radians, R^T is a rotation matrix about the same axis that rotates a vector by $-\theta$ radians. If R is a reflection matrix through a plane $\mathbf{N} \cdot \mathbf{X} = 0$, then $R^T = R$, which says that if you reflect twice, you end up where you started.

One distinguishing algebraic characteristic between rotations and reflections is the value of their determinants. Recall that the determinant of a 3×3 matrix M is

$$\begin{aligned} \det(M) &= \det \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} \\ &= m_{00}m_{11}m_{22} + m_{01}m_{12}m_{20} + m_{02}m_{10}m_{21} \\ &\quad - m_{02}m_{11}m_{20} - m_{01}m_{10}m_{22} - m_{00}m_{12}m_{21} \end{aligned} \quad (2.24)$$

The determinant of a rotation matrix is 1. The determinant of a reflection matrix is -1 .

EXERCISE 2.4

Verify that the determinant of the matrix in Equation (2.22) is 1. Verify that the determinant of the matrix in Equation (2.23) is -1 . ■

Scaling

Scaling is a simple transformation. You scale each component of a vector by a desired amount: $(y_0, y_1, y_2) = (s_0x_0, s_1x_1, s_2x_2)$, where s_0 , s_1 , and s_2 are the scaling factors. The matrix that represents scaling is

$$S = \begin{bmatrix} s_0 & 0 & 0 \\ 0 & s_1 & 0 \\ 0 & 0 & s_2 \end{bmatrix} \quad (2.25)$$

This is a diagonal matrix. Usually, we assume that the scaling factors are positive numbers, but 3D modeling packages tend to allow you to set them to negative numbers, which can cause all sorts of problems for exporters and engines that rely on positive scales. If the scales are all the same value, $s_0 = s_1 = s_2$, the transformation is said to be a *uniform scaling*; otherwise, it is a *nonuniform scaling*. Nonuniform scales can also lead to problems in the design of a graphics engine. I will get into the details of this later in the design of a scene graph hierarchy for which each node stores rotation and scaling matrices and translation vectors but also stores a composite matrix for the entire transformation. Given a composite matrix, a frequently asked question is how to extract the rotational component and the scaling factors. We will see that this is an ill-posed problem; see Section 17.5.

The scaling matrix of Equation (2.25) represents scaling in the directions of the coordinate axes. It is possible to scale in different directions. For example, if you want to scale the vectors by s in the direction $\mathbf{D}$, you need to decompose the input point $\mathbf{X}$ into a $\mathbf{D}$ component and a remainder:

$$\mathbf{X} = d\mathbf{D} + \mathbf{R}$$

where $\mathbf{R}$ is perpendicular to $\mathbf{D}$. The decomposition is similar to what was used to construct reflections. The component of $\mathbf{X}$ in the $\mathbf{D}$ direction is $d\mathbf{X}$, where $d = \mathbf{D} \cdot \mathbf{X}$. The vector scaled in the $\mathbf{D}$ direction is

$$\mathbf{Y} = sd\mathbf{D} + \mathbf{R}$$

and is illustrated in Figure 2.6.

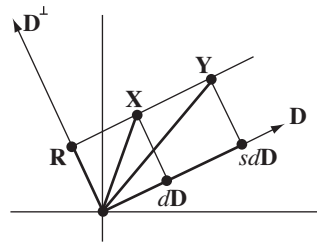


Figure 2.6 Scaling of a vector $\mathbf{X}$ by a scaling factor s in the direction $\mathbf{D}$ to produce a vector $\mathbf{Y}$.

Some algebra will show that

$$\mathbf{Y} = \left(s\mathbf{D}\mathbf{D}^T + \mathbf{D}^\perp (\mathbf{D}^\perp)^T \right) \mathbf{X}$$

where $\mathbf{D}^\perp$ is a unit-length vector perpendicular to the unit-length vector $\mathbf{D}$.

Generally, in three dimensions you can apply scaling in three noncoordinate axis directions $\mathbf{D}$, $\mathbf{U}$, and $\mathbf{R}$ by representing the input point in this new coordinate system as

$$\mathbf{X} = d\mathbf{D} + u\mathbf{U} + r\mathbf{R}$$

and then scaling each of the components:

$$\mathbf{Y} = s_0 d\mathbf{D} + s_1 u\mathbf{U} + s_2 r\mathbf{R}$$

In matrix form, this becomes

$$\mathbf{Y} = \left(s_0\mathbf{D}\mathbf{D}^T + s_1\mathbf{U}\mathbf{U}^T + s_2\mathbf{R}\mathbf{R}^T \right) \mathbf{X} \quad (2.26)$$

EXERCISE 2.5

Construct Equation (2.26). *Hint:* Use the fact that $d = \mathbf{D} \cdot \mathbf{X}$, $u = \mathbf{U} \cdot \mathbf{X}$, and $r = \mathbf{R} \cdot \mathbf{X}$. Use the construction that led to Equation (2.23) to help you decide how to rearrange the various vector terms appropriately. ■

A more intuitive equation for general scaling than Equation (2.26) is obtained by using a matrix $M = [\mathbf{D} \ \mathbf{U} \ \mathbf{R}]$. The notation means that the first column of M is the 3×1 vector $\mathbf{D}$, the second column is the 3×1 vector $\mathbf{U}$, and the third column is the 3×1 vector $\mathbf{R}$. The transpose of M is also written in a concise form,

$$M^T = \begin{bmatrix} \mathbf{D}^T \\ \mathbf{U}^T \\ \mathbf{R}^T \end{bmatrix}$$

The notation means that the first row of M is the 1×3 vector $\mathbf{D}^T$, the second row of M is the 1×3 vector $\mathbf{U}^T$, and the third row of M is the 1×3 vector $\mathbf{R}^T$. Let S be the diagonal matrix of Equation (2.25). The scaling matrix becomes

$$s_0\mathbf{D}\mathbf{D}^T + s_1\mathbf{U}\mathbf{U}^T + s_2\mathbf{R}\mathbf{R}^T = [\mathbf{D} \ \mathbf{U} \ \mathbf{R}] \begin{bmatrix} s_0 & 0 & 0 \\ 0 & s_1 & 0 \\ 0 & 0 & s_2 \end{bmatrix} \begin{bmatrix} \mathbf{D}^T \\ \mathbf{U}^T \\ \mathbf{R}^T \end{bmatrix} = M S M^T \quad (2.27)$$

Both M and M^T are rotation matrices since we are assuming that our coordinate systems have direction vectors that are mutually perpendicular and form a right-handed system. In words, M^T rotates $\mathbf{X}$ to the new coordinate system, S scales the rotated vector, and M rotates the result back to the old coordinate system.

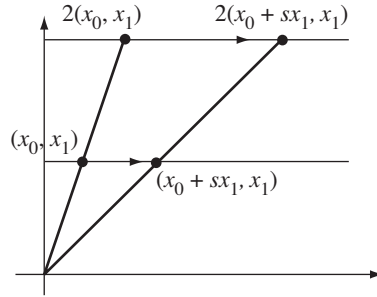


Figure 2.7 Shearing of points (x_0, x_1) in the x_0 direction. As the x_1 value increases for points, the amount of shearing in the x_0 direction increases.

Shearing

Shearing operations are applied less often than rotations, reflections, and scalings, but I include them here anyway since they tend to be grouped into those transformations of interest in computer graphics. To motivate the idea, consider a shearing in two dimensions, as illustrated in Figure 2.7.

In two dimensions, the shearing in the x_0 direction has the matrix representation

$$S = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}$$

This maps the point (x_0, x_1) to $(y_0, y_1) = (x_0 + sx_1, x_1)$, as shown in Figure 2.7. If you want to shear in the x_1 direction, the matrix is similar,

$$S = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$$

This maps the point (x_0, x_1) to $(y_0, y_1) = (x_0, x_1 + sx_0)$.

In three dimensions, shearing is applied within planes. For example, if you want to shear within the planes parallel to the x_0x_1 plane, you would use the matrix

$$S_{01} = \begin{bmatrix} 1 & s & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.28)$$

to shear within each plane $x_2 = c$ (constant) in the direction of x_0 . Notice that (x_0, x_1, x_2) is mapped to $(x_0 + sx_1, x_1, x_2)$, so only the x_0 value is changed by the

shearing. If you want to shear within these planes, but in the x_1 direction, you would use the matrix

$$S_{10} = \begin{bmatrix} 1 & 0 & 0 \\ s & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.29)$$

In the two equations, the subscripts on the matrix names refer to the indices of the matrix entries that contain the shearing factors.

Similarly, you can shear in the planes $x_1 = c$ (constant) in the x_0 direction by using the matrix

$$S_{02} = \begin{bmatrix} 1 & 0 & s \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.30)$$

or you can shear in the x_2 direction by using the matrix

$$S_{20} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ s & 0 & 1 \end{bmatrix} \quad (2.31)$$

You can shear in the planes $x_0 = c$ (constant) in the x_1 direction by using the matrix

$$S_{12} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & s \\ 0 & 0 & 1 \end{bmatrix} \quad (2.32)$$

or you can shear in the x_2 direction by using the matrix

$$S_{21} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & s & 1 \end{bmatrix} \quad (2.33)$$

Shearing in an arbitrary plane containing the origin is possible. The construction of the matrix is similar to what was done for scaling; see Equation (2.27) and the discussion leading to it. You rotate the input point $\mathbf{X}$ to the coordinate system of interest, apply one of the coordinate scaling transformations, and then rotate the result back to the original coordinate system. The general transformation is

$$[\mathbf{D} \quad \mathbf{U} \quad \mathbf{R}] S_{ij} \begin{bmatrix} \mathbf{D}^T \\ \mathbf{U}^T \\ \mathbf{R}^T \end{bmatrix} \quad (2.34)$$

where S_{ij} is one of the shearing matrices from Equations (2.28) through (2.33).

2.2.2 AFFINE TRANSFORMATIONS

As noted previously, translation is not a linear transformation. However, it is what distinguishes an *affine transformation* from a linear one. The definition of a linear transformation says that a transformation of a linear combination is a linear combination of the transformations. In mathematical terms, if c_i are scalars and $\mathbf{V}_i$ are vectors for $1 \leq i \leq n$, and if $\mathbf{L}$ is a linear transformation, then

$$\mathbf{L} \left(\sum_{i=1}^n c_i \mathbf{V}_i \right) = \sum_{i=1}^n c_i \mathbf{L}(\mathbf{V}_i)$$

An affine transformation must deal with the distinction between points and vectors. For the moment, I will switch back to the typesetting conventions for points and vectors. Let $\mathcal{P}$ and $\mathcal{Q}$ be points. Let $\mathcal{Q} = \mathcal{A}(\mathcal{P})$ be a transformation that maps points to points. The transformation is an *affine transformation* when it satisfies the following conditions:

1. Consider points $\mathcal{P}_i$ and $\mathcal{Q}_i = \mathcal{A}(\mathcal{P}_i)$ for $1 \leq i \leq 4$. If $\mathcal{P}_2 - \mathcal{P}_1 = \mathcal{P}_4 - \mathcal{P}_3$, then it must be that $\mathcal{Q}_2 - \mathcal{Q}_1 = \mathcal{Q}_4 - \mathcal{Q}_3$.
2. If $\mathbf{X} = \mathcal{P}_2 - \mathcal{P}_1$ and $\mathbf{Y} = \mathcal{Q}_2 - \mathcal{Q}_1$, then the transformation $\mathbf{Y} = \mathbf{L}(\mathbf{X})$ must be a linear transformation.

This is a fancy mathematical way of saying that an affine transformation is composed of two parts, one part that maps a point to a point and one part that maps a vector to a vector. Suppose that the linear transformation is written in matrix form, $\mathbf{Y} = \mathbf{M}\mathbf{X}$, for some matrix $\mathbf{M}$. The second condition in the definition implies

$$\mathcal{P}_2 = \mathcal{P}_1 + \mathbf{X} \tag{2.35}$$

and

$$\mathcal{Q}_2 = \mathcal{Q}_1 + \mathbf{Y} = \mathcal{Q}_1 + \mathbf{M}\mathbf{X}$$

Since $\mathcal{Q}_1$ is a point, we can write it as an offset from $\mathcal{P}_1$; namely,

$$\mathcal{Q}_1 = \mathcal{P}_1 + \mathbf{B}$$

for some vector $\mathbf{B}$. Combining the last two displayed equations, we have

$$\mathcal{Q}_2 = \mathcal{P}_1 + (\mathbf{M}\mathbf{X} + \mathbf{B}) \tag{2.36}$$

The affine transformation between points is

$$\mathcal{Q}_2 = \mathcal{A}(\mathcal{P}_2)$$

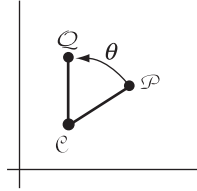


Figure 2.8 Rotation of a point $\mathcal{P}$ about a central point $\mathcal{C}$ to obtain a point $\mathcal{Q}$.

but Equation (2.36) says that as long as we use the same origin for the coordinate space, we may compute the transformation in vector terms:

$$\mathbf{Y} = \mathbf{M}\mathbf{X} + \mathbf{B} \quad (2.37)$$

You should recognize this as the standard form in which you have manipulated vectors when translations are allowed. It is important to note that the right-hand side of Equation (2.37) consists only of vector and matrix operations. There are no “point” operations. Perhaps this is yet another argument why your graphics engine need not enforce a distinction between points and vectors.

Although translation is the obvious candidate to illustrate affine transformations, another one of interest is rotation about a point that is not at the origin, as Figure 2.8 illustrates.

The origin of the coordinate system is $\mathcal{O} = (0, 0, 0)$. The vector from the origin to the center of rotation is defined by $\mathbf{C} = \mathcal{C} - \mathcal{O}$. The rotation matrix is R . The points are $\mathcal{P} = \mathcal{O} + \mathbf{X}$ and

$$\begin{aligned} \mathcal{Q} &= \mathcal{O} + \mathbf{Y} \\ &= \mathcal{C} + (\mathcal{O} - \mathcal{C}) + \mathbf{Y} \\ &= \mathcal{C} + (\mathbf{Y} - \mathbf{C}) \\ &= \mathcal{C} + R(\mathbf{X} - \mathbf{C}) \\ &= \mathcal{O} + (\mathcal{C} - \mathcal{O}) + R(\mathbf{X} - \mathbf{C}) \\ &= \mathcal{O} + \mathbf{C} + R(\mathbf{X} - \mathbf{C}) \end{aligned}$$

Removing the point notation, the vector calculations are

$$\mathbf{Y} = R\mathbf{X} + (I - R)\mathbf{C} \quad (2.38)$$

where I is the identity matrix. The translational component of this representation for the affine transformation is $(I - R)\mathbf{C}$.

Note: For the remainder of the book, I will use the same typesetting convention for points and vectors. When necessary, I will state explicitly whether something is a point or a vector.

2.2.3 PROJECTIVE TRANSFORMATIONS

Yet another class of transformations involve *projections*. There are different types of projections that will interest us: orthogonal, oblique, and perspective.

Orthogonal Projection onto a Line

Orthogonal projection is the simplest type of projection to analyze. Consider the goal of projecting a point onto a line. Figure 2.9 illustrates this.

The point $\mathbf{X}$ is to be projected onto a line containing a point $\mathbf{P}$ and having unit-length direction $\mathbf{D}$. In the figure, the point $\mathbf{Y}$ is the projection and has the property that the vector $\mathbf{X} - \mathbf{Y}$ is perpendicular to $\mathbf{D}$; that is,

$$0 = \mathbf{D} \cdot (\mathbf{X} - \mathbf{Y})$$

Because $\mathbf{Y}$ is on the line, it is of the form

$$\mathbf{Y} = \mathbf{P} + d\mathbf{D}$$

for some scalar d . Substituting this in the previous displayed equation, we have

$$0 = \mathbf{D} \cdot (\mathbf{X} - \mathbf{P} - d\mathbf{D})$$

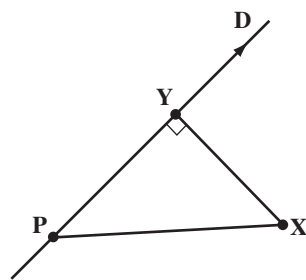


Figure 2.9 The orthogonal projection of a point onto a line.

which can be solved to obtain

$$d = \mathbf{D} \cdot (\mathbf{X} - \mathbf{P})$$

The point of projection is therefore defined by

$$\mathbf{Y} - \mathbf{P} = (\mathbf{D} \cdot (\mathbf{X} - \mathbf{P}))\mathbf{D} = \mathbf{D}\mathbf{D}^T(\mathbf{X} - \mathbf{P})$$

Equivalently, the projection is

$$\mathbf{Y} = \mathbf{D}\mathbf{D}^T\mathbf{X} + \left(\mathbf{I} - \mathbf{D}\mathbf{D}^T\right)\mathbf{P} \quad (2.39)$$

which is of the form $\mathbf{Y} = \mathbf{M}\mathbf{X} + \mathbf{B}$, therefore orthogonal projection onto a line is an affine transformation. Unlike our previous examples, this transformation is not invertible. Each point on a line has infinitely many points that project to it (an entire plane's worth), so you cannot unproject a point from the line unless you have more information. Algebraically, the noninvertibility shows up in that $\mathbf{M} = \mathbf{D}\mathbf{D}^T$ is not an invertible matrix.

Orthogonal Projection onto a Plane

Consider projecting a point $\mathbf{X}$ onto a plane defined by $\mathbf{N} \cdot (\mathbf{Y} - \mathbf{P}) = 0$, where $\mathbf{N}$ is a unit-length normal vector, $\mathbf{P}$ is a specified point, and $\mathbf{Y}$ is any point on the plane. Figure 2.10 illustrates this.

The projection point is $\mathbf{Y}$. The vector $\mathbf{X} - \mathbf{P}$ has a component in the plane, $\mathbf{Y} - \mathbf{P}$, and a component on the normal line to the plane, $n\mathbf{N}$, for some scalar n . Thus,

$$\mathbf{X} - \mathbf{P} = (\mathbf{Y} - \mathbf{P}) + n\mathbf{N}$$

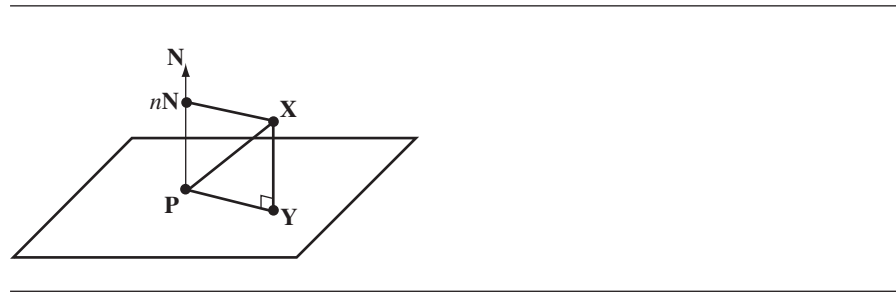


Figure 2.10 The orthogonal projection of a point onto a plane.

Dotting with $\mathbf{N}$, we have

$$\mathbf{N} \cdot (\mathbf{X} - \mathbf{P}) = n$$

which uses the facts that $\mathbf{N}$ is unit length and $\mathbf{Y} - \mathbf{P}$ is perpendicular to $\mathbf{N}$. We may solve to obtain

$$\mathbf{Y} - \mathbf{P} = (\mathbf{X} - \mathbf{P}) - (\mathbf{N} \cdot (\mathbf{X} - \mathbf{P}))\mathbf{N} = (\mathbf{I} - \mathbf{N}\mathbf{N}^T)(\mathbf{X} - \mathbf{P})$$

Equivalently, the projection is

$$\mathbf{Y} = (\mathbf{I} - \mathbf{N}\mathbf{N}^T)\mathbf{X} + \mathbf{N}\mathbf{N}^T\mathbf{P} \quad (2.40)$$

You will notice that this is of the form $\mathbf{Y} = \mathbf{M}\mathbf{X} + \mathbf{B}$, therefore orthogonal projection onto a plane is also an affine transformation. Moreover, it is not invertible since infinitely many points in space project to the same point on the plane (an entire line's worth), so you cannot unproject a point from the plane unless you have more information. Algebraically, the noninvertibility shows up in that $\mathbf{M} = \mathbf{I} - \mathbf{N}\mathbf{N}^T$ is not an invertible matrix.

Oblique Projection onto a Plane

As before, let the plane contain a point $\mathbf{P}$ and have a unit-length normal vector $\mathbf{N}$. The projection of a point onto a plane does not have to be in the normal direction to the plane. This type of projection is said to be *oblique* to the plane. Let $\mathbf{D}$ be the unit-length direction in which to project the points. This direction should not be perpendicular to the plane; that is, $\mathbf{N} \cdot \mathbf{D} \neq 0$. Figure 2.11 illustrates this.

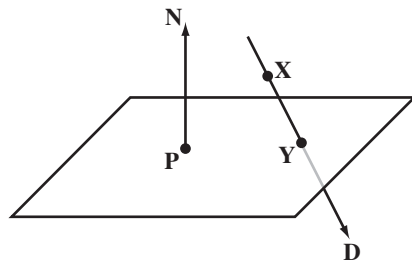


Figure 2.11 The oblique projection of a point onto a plane.

The projection point is $\mathbf{Y} = \mathbf{X} + d\mathbf{D}$ for some scalar d . Subtracting the known plane point, we have

$$\mathbf{Y} - \mathbf{P} = (\mathbf{X} - \mathbf{P}) + d\mathbf{D}$$

Dotting with $\mathbf{N}$, we have

$$0 = \mathbf{N} \cdot (\mathbf{X} - \mathbf{P}) + d\mathbf{N} \cdot \mathbf{D}$$

which uses the fact that $\mathbf{Y} - \mathbf{P}$ is perpendicular to $\mathbf{N}$. We may solve for d ,

$$d = -\frac{\mathbf{N} \cdot (\mathbf{X} - \mathbf{P})}{\mathbf{N} \cdot \mathbf{D}}$$

The projection is defined, therefore, by

$$\mathbf{Y} - \mathbf{P} = (\mathbf{X} - \mathbf{P}) - \frac{\mathbf{N} \cdot (\mathbf{X} - \mathbf{P})}{\mathbf{N} \cdot \mathbf{D}} \mathbf{D} = \left(I - \frac{\mathbf{D}\mathbf{N}^T}{\mathbf{D}^T\mathbf{N}} \right) (\mathbf{X} - \mathbf{P})$$

Equivalently, the projection is

$$\mathbf{Y} = \left(I - \frac{\mathbf{D}\mathbf{N}^T}{\mathbf{D}^T\mathbf{N}} \right) \mathbf{X} + \frac{\mathbf{D}\mathbf{N}^T}{\mathbf{D}^T\mathbf{N}} \mathbf{P} \quad (2.41)$$

Once again, this is of the form $M\mathbf{X} + \mathbf{B}$, therefore oblique projection onto a plane is an affine transformation. And as with the other projections, it is not invertible.

Perspective Projection onto a Plane

We now encounter a projection that is not an affine transformation, and one that is central to rendering—the perspective projection. Points are now projected onto a plane, but along rays with a common origin $\mathbf{E}$, called the *eye point*. Figure 2.12 illustrates this.

The point $\mathbf{X}$ is projected to the point $\mathbf{Y}$. The ray to use has origin $\mathbf{E}$, but the direction is determined by the vector $\mathbf{X} - \mathbf{E}$. At the moment there is no need to use a unit-length vector for the direction. As a ray point, we have

$$\mathbf{Y} = \mathbf{E} + t(\mathbf{X} - \mathbf{E})$$

for some scalar $t > 0$. Subtract the known plane point $\mathbf{P}$ to obtain

$$\mathbf{Y} - \mathbf{P} = (\mathbf{E} - \mathbf{P}) + t(\mathbf{X} - \mathbf{E})$$

and dot with $\mathbf{N}$ to obtain

$$0 = \mathbf{N} \cdot (\mathbf{E} - \mathbf{P}) + t\mathbf{N} \cdot (\mathbf{X} - \mathbf{E})$$

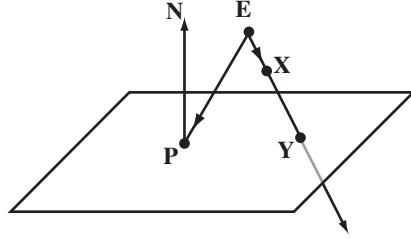


Figure 2.12 The perspective projection of a point onto a plane using an eye point E .

This uses the fact that $Y - P$ is perpendicular to the plane normal N . Solving for t , we have

$$t = -\frac{N \cdot (E - P)}{N \cdot (X - E)}$$

To reaffirm the constraint that $t \geq 0$, Figure 2.12 shows the vectors $P - E$ and $X - E$. Both vectors form an obtuse angle with the normal vector, so $N \cdot (P - E) < 0$ and $N \cdot (X - E) < 0$, which imply $t > 0$.

The projection point is, therefore, defined by

$$Y = E - \frac{N \cdot (E - P)}{N \cdot (X - E)} (X - E) = \frac{(EN^T - N \cdot (E - P)I) (X - E)}{N \cdot (X - E)} \quad (2.42)$$

where I is the identity matrix. It is not possible to write the expression in the form $Y = MX + B$, where M and B are independent of X . What prevents this is the division by $N \cdot (X - E)$. You will hear reference to this division as the *perspective divide*. However, Section 2.2.5 will show a unifying format for representing linear, affine, and perspective transformations.

EXERCISE 2.6

The point X is “behind the eye” when the distance from X to the plane is larger than or equal to the distance from E to the plane. In this case, prove that the ray from E to X cannot intersect the plane. ■

2.2.4 PROPERTIES OF PERSPECTIVE PROJECTION

Consider the coordinate system whose origin is the eye point, whose view direction is D , whose up direction is U , and whose right direction is $R = D \times U$. The point to be projected has a representation in this coordinate system as

$$X = E + dD + uU + rR$$

where $d = \mathbf{D} \cdot (\mathbf{X} - \mathbf{E})$, $u = \mathbf{D} \cdot (\mathbf{X} - \mathbf{E})$, and $r = \mathbf{D} \cdot (\mathbf{X} - \mathbf{E})$. The plane normal is in the opposite direction of the view; namely, $\mathbf{N} = -\mathbf{D}$. The point on the plane closest to the eye point is $\mathbf{P} = \mathbf{E} + d_{\min} \mathbf{D}$ for some distance $d_{\min} > 0$. Substituting all this into Equation (2.42), we have the projected point

$$\mathbf{Y} = \mathbf{E} + \frac{d_{\min}(d\mathbf{D} + u\mathbf{U} + r\mathbf{R})}{d} = \mathbf{E} + \frac{d\mathbf{D} + u\mathbf{U} + r\mathbf{R}}{d/d_{\min}}$$

Thus, in the new coordinate system, (d, u, r) is projected to $(d, u, r)/(d/d_{\min})$. Notice that the first component is actually $d_{\min}$, which is to be expected since the projection point is on the plane $d = d_{\min}$ (in the new coordinate system).

Within the new coordinate system, it is relatively easy to demonstrate some properties of perspective projection. In all of the cases mentioned here, the objects are assumed to be in front of the eye point; that is, all points on the objects are closer to the projection plane than the eye point. Line segments must project to line segments, or to a single point when the line segment is fully contained by a single ray emanating from $\mathbf{E}$. Triangles must project to triangles, or to a line segment when the triangle and $\mathbf{E}$ are coplanar. Finally, conic sections project to conic sections, with the degenerate case of a conic section projecting to linear components when the conic section is coplanar with $\mathbf{E}$. A consequence of verifying these properties is that we will have an idea of how uniformly spaced points on a line segment are projected to nonuniformly spaced points. This relationship is important in perspective projection for rendering, in particular when depth must be computed (which is nearly always) and in interpolating vertex attributes.

Lines Project to Lines

Consider a line segment with endpoints $\mathbf{Q}_i = (d_i, u_i, r_i)$ for $i = 0, 1$. Using only the two components relevant to the projection plane, let the corresponding projected points be $\mathbf{P}_i = (u_i/w_i, r_i/w_i)$, with $w_i = d_i/d_{\min}$ for $i = 0, 1$. The 3D line segment is $\mathbf{Q}(s) = \mathbf{Q}_0 + s(\mathbf{Q}_1 - \mathbf{Q}_0)$ for $s \in [0, 1]$. For each s , let $\mathbf{P}(s)$ be the projection of $\mathbf{Q}(s)$. Thus,

$$\mathbf{Q}(s) = (d_0 + s(d_1 - d_0), u_0 + s(u_1 - u_0), r_0 + s(r_1 - r_0))$$

and

$$\begin{aligned} \mathbf{P}(s) &= \left(\frac{u_0 + s(u_1 - u_0)}{w_0 + s(w_1 - w_0)}, \frac{r_0 + s(r_1 - r_0)}{w_0 + s(w_1 - w_0)} \right) \\ &= \left(\frac{u_0}{w_0} + \frac{w_1 s}{w_0 + (w_1 - w_0)s} \left(\frac{u_1}{w_1} - \frac{u_0}{w_0} \right), \right. \\ &\quad \left. \frac{r_0}{w_0} + \frac{w_1 s}{w_0 + (w_1 - w_0)s} \left(\frac{r_1}{w_1} - \frac{r_0}{w_0} \right) \right) \end{aligned}$$

$$\begin{aligned}
&= \mathbf{P}_0 + \frac{w_1 s}{w_0 + (w_1 - w_0)s} (\mathbf{P}_1 - \mathbf{P}_0) \\
&= \mathbf{P}_0 + \bar{s} (\mathbf{P}_1 - \mathbf{P}_0)
\end{aligned}$$

where the last equality defines

$$\bar{s} = \frac{w_1 s}{w_0 + (w_1 - w_0)s} \quad (2.43)$$

a quantity that is also in the interval $[0, 1]$. We have obtained a parametric equation for a 2D line segment with endpoints $\mathbf{P}_0$ and $\mathbf{P}_1$, so in fact line segments are projected to line segments, or if $\mathbf{P}_0 = \mathbf{P}_1$, the projected segment is a single point. The inverse mapping from $\bar{s}$ to s is actually important for perspective correct rasterization, as we will see later:

$$s = \frac{w_0 \bar{s}}{w_1 + (w_0 - w_1)\bar{s}} \quad (2.44)$$

EXERCISE 2.7

Construct Equation (2.44) from Equation (2.43). ■

Equation (2.43) has more to say about perspective projection. Assuming $w_1 > w_0$, a uniform change in s does not result in a uniform change in $\bar{s}$. The graph of $\bar{s} = F(s)$ is shown in Figure 2.13.

The first derivative is $F'(s) = w_0 w_1 / [w_0 + s(w_1 - w_0)]^2 > 0$, and the second derivative is $F''(s) = -2w_0 w_1 / [w_0 + s(w_1 - w_0)]^3 < 0$. The slopes of the graph at the endpoints are $F'(0) = w_1/w_0 > 1$ and $F'(1) = w_0/w_1 < 1$. Since the second derivative is always negative, the graph is concave. An intuitive interpretation is to select a set of uniformly spaced points on the 3D line segment. The projections of these

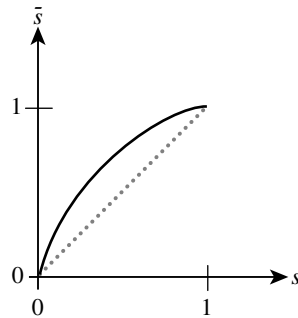


Figure 2.13 The relationship between s and $\bar{s}$.

points are not uniformly spaced. More specifically, the spacing between the projected points decreases as $\bar{s}$ increases from 0 to 1. The relationship between s and $\bar{s}$ and limited floating-point precision are what contribute to depth buffering artifacts, to be discussed later.

Triangles Project to Triangles

Because line segments project to line segments, we can immediately assert that triangles project to triangles, although possibly degenerating to a line segment. However, let's derive the parametric relationships that are analogous to those of Equations (2.43) and (2.44) anyway.

Let $\mathbf{Q}_i = (d_i, u_i, r_i)$ for $i = 0, 1, 2$ be the vertices of a triangle. The triangle is specified parametrically as $\mathbf{Q}(s, t) = \mathbf{Q}_0 + s(\mathbf{Q}_1 - \mathbf{Q}_0) + t(\mathbf{Q}_2 - \mathbf{Q}_0)$ for $0 \leq s \leq 1$, $0 \leq t \leq 1$, and $s + t \leq 1$. Let the projected points for the $\mathbf{Q}_i$ be $\mathbf{P}_i = (u_i/w_i, r_i/w_i)$ for $i = 0, 1, 2$, where $w_i = d_i/d_{\min}$. For each s and t , let $\mathbf{P}(s, t)$ be the projection of $\mathbf{Q}(s, t)$. Some algebra will show the following, where $\Delta = w_0 + (w_1 - w_0)s + (w_2 - w_0)t$,

$$\begin{aligned} \mathbf{P}(s, t) &= \left(\frac{u_0 + s(u_1 - u_0) + t(u_2 - u_0)}{\Delta}, \frac{r_0 + s(r_1 - r_0) + t(r_2 - r_0)}{\Delta} \right) \\ &= \left(\frac{u_0}{w_0} + \frac{w_1 s}{\Delta} \left(\frac{u_1}{w_1} - \frac{u_0}{w_0} \right) + \frac{w_2 t}{\Delta} \left(\frac{u_2}{w_2} - \frac{u_0}{w_0} \right), \frac{r_0}{w_0} + \frac{w_1 s}{\Delta} \left(\frac{r_1}{w_1} - \frac{r_0}{w_0} \right) \right. \\ &\quad \left. + \frac{w_2 t}{\Delta} \left(\frac{r_2}{w_2} - \frac{r_0}{w_0} \right) \right) \\ &= \mathbf{P}_0 + \frac{w_1 s}{\Delta} (\mathbf{P}_1 - \mathbf{P}_0) + \frac{w_2 t}{\Delta} (\mathbf{P}_2 - \mathbf{P}_0) \end{aligned}$$

Define

$$(\bar{s}, \bar{t}) = \frac{(w_1 s, w_2 t)}{w_0 + (w_1 - w_0)s + (w_2 - w_0)t} \quad (2.45)$$

in which case the projected triangle is

$$\mathbf{P}(s, t) = \mathbf{P}_0 + \bar{s}(\mathbf{P}_1 - \mathbf{P}_0) + \bar{t}(\mathbf{P}_2 - \mathbf{P}_0)$$

The inverse mapping can be used by the rasterizers for perspective correct triangle rasterization. The inverse is

$$(s, t) = \frac{(w_0 w_2 \bar{s}, w_0 w_1 \bar{t})}{w_1 w_2 + w_2(w_0 - w_1)\bar{s} + w_1(w_0 - w_2)\bar{t}} \quad (2.46)$$

EXERCISE
2.8

Construct Equation (2.46) from Equation (2.45). ■

Conics Project to Conics

Showing that the projection of a conic section is itself a conic section requires a bit more algebra. Let $\mathbf{Q}_i = (x_i, y_i, z_i)$ for $i = 0, 1, 2$ be points such that $\mathbf{Q}_1 - \mathbf{Q}_0$ and $\mathbf{Q}_2 - \mathbf{Q}_0$ are unit length and orthogonal. The points in the plane containing the $\mathbf{Q}_i$ are represented by $\mathbf{Q}(s, t) = \mathbf{Q}_0 + s(\mathbf{Q}_1 - \mathbf{Q}_0) + t(\mathbf{Q}_2 - \mathbf{Q}_0)$ for any real numbers s and t . Within that plane, a conic section is defined by

$$As^2 + Bst + Ct^2 + Ds + Et + F = 0 \quad (2.47)$$

To show that the projection is also a conic, substitute the formulas in Equation (2.46) into Equation (2.47) to obtain

$$\bar{A}\bar{s}^2 + \bar{B}\bar{s}\bar{t} + \bar{C}\bar{t}^2 + \bar{D}\bar{s} + \bar{E}\bar{t} + \bar{F} = 0 \quad (2.48)$$

where

$$\bar{A} = w_2^2 (w_0^2 A + w_0(w_0 - w_1)D + (w_0 - w_1)^2 F)$$

$$\bar{B} = w_1 w_2 (w_0^2 B + w_0(w_0 - w_2)D + w_0(w_0 - w_1)E + 2(w_0 - w_1)(w_0 - w_2)F)$$

$$\bar{C} = w_1^2 (w_0^2 C + w_0(w_0 - w_2)E + (w_0 - w_2)^2 F)$$

$$\bar{D} = w_1 w_2^2 (w_0 D + 2(w_0 - w_1)F)$$

$$\bar{E} = w_1^2 w_2 (w_0 E + 2(w_0 - w_2)F)$$

$$\bar{F} = w_1^2 w_2^2 F.$$

A special case is $D = E = F = 0$, in which case the conic is centered at $\mathbf{Q}_0$ and has axes $\mathbf{Q}_1 - \mathbf{Q}_0$ and $\mathbf{Q}_2 - \mathbf{Q}_0$. Consequently, $\bar{A} = w_2^2 w_0^2 A$, $\bar{B} = w_1 w_2 w_0^2 B$, $\bar{C} = w_1^2 w_0^2 C$, and $\bar{B}^2 - 4\bar{A}\bar{C} = B^2 - 4AC$. The sign of $B^2 - 4AC$ is preserved, so ellipses are mapped to ellipses, hyperbolas are mapped to hyperbolas, and parabolas are mapped to parabolas.

EXERCISE
2.9

At the beginning of the discussion about projecting line segments, triangles, and conic sections, the assumption was made that all points on these objects are in front of the eye point. What can you say about the projections of the objects when some points are behind the eye point? ■

2.2.5 HOMOGENEOUS POINTS AND MATRICES

We have seen that linear transformations are of the form

$$\mathbf{Y} = M\mathbf{X}$$

where $\mathbf{X}$ is the 3×1 input vector, $\mathbf{Y}$ is the 3×1 output vector, and M is a 3×3 matrix of constants. Affine transformations extend this to

$$\mathbf{Y} = M\mathbf{X} + \mathbf{B}$$

where $\mathbf{B}$ is a 3×1 vector of constants. The perspective transformation did not fit within this framework. Equation (2.42) is of the form

$$\mathbf{Y} = \frac{M(\mathbf{X} - \mathbf{E})}{\mathbf{N} \cdot (\mathbf{X} - \mathbf{E})}$$

We can unify these into a single matrix representation by introducing the concept of *homogeneous points*, which are represented as 4-tuples but written as 4×1 vectors when used in matrix-vector operations, and *homogeneous matrices*, which are represented as 4×4 matrices.

Using the standard naming conventions that graphics practitioners have used for homogeneous points, the 4-tuples are of the form (x, y, z, w) . I had already hinted at using 4-tuples to distinguish between points and vectors; see Section 2.1.3. The 4-tuple represents a point when $w = 1$, so $(x, y, z, 1)$ is a point. The 4-tuple represents a vector when $w = 0$, so $(x, y, z, 0)$ is a vector. In computer graphics, though, there is more to homogeneous points than specifying w to be zero or one.

Defining points as 4-tuples already allows us to unify linear and affine transformations into a single matrix representation. Specifically, the linear transformation is currently of the form

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \mathbf{Y} = M\mathbf{X} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

Appending a fourth component of 1 to the vectors and increasing the size of the matrix, adding certain entries as needed, this equation becomes

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ 1 \end{bmatrix}$$

A convenient *block-matrix form* is

$$\begin{bmatrix} \mathbf{Y} \\ 1 \end{bmatrix} = \begin{bmatrix} M & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix}$$

The output vector has an upper block that is the 3×1 vector $\mathbf{Y}$. The lower block is the (1×1) scalar 1. The input vector is structured similarly. The matrix of coefficients has the following structure. The upper-left block is the 3×3 matrix M ; the upper-right block is the 3×1 zero vector; the lower-left block is the 1×3 zero vector; and the lower-right block is the (1×1) scalar 1.

Similarly, the affine transformation currently is

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \mathbf{Y} = M\mathbf{X} + \mathbf{B} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

but may be extended to use 4×1 points and a 4×4 matrix,

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & b_0 \\ m_{10} & m_{11} & m_{12} & b_1 \\ m_{20} & m_{21} & m_{22} & b_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ 1 \end{bmatrix}$$

It also has a block-matrix form,

$$\begin{bmatrix} \mathbf{Y} \\ 1 \end{bmatrix} = \begin{bmatrix} M & \mathbf{B} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix}$$

The perspective transformation can *almost* be made to fit into this framework, with two notable exceptions. First, the choice of $w = 1$ for the input is acceptable, but the output value for w is generally nonzero and not 1. Second, we still cannot handle the perspective divide. Despite this, consider the following block-matrix expression that gets us closer to our goal:

$$\begin{bmatrix} \mathbf{Y} \\ 1 \end{bmatrix} \sim \begin{bmatrix} \mathbf{Y}' \\ w \end{bmatrix} = \begin{bmatrix} M(\mathbf{X} - \mathbf{E}) \\ \mathbf{N}^T(\mathbf{X} - \mathbf{E}) \end{bmatrix} = \begin{bmatrix} M & -M\mathbf{E} \\ \mathbf{N}^T & -\mathbf{N}^T\mathbf{E} \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} \quad (2.49)$$

where $M = \mathbf{E}\mathbf{N}^T - \mathbf{N} \cdot (\mathbf{E} - \mathbf{P})I$. The leftmost expression is what we want to construct, where $\mathbf{Y}$ is the output vector defined by Equation (2.42). The remaining portions of the expression are what we can construct using matrix operations. The output of these operations has a 3×1 vector $\mathbf{Y}'$, which is not the actual output we want. The output also has a w component, which is not necessarily 1. However, if we were to perform the perspective divide, we obtain

$$\mathbf{Y} = \mathbf{Y}'/w$$

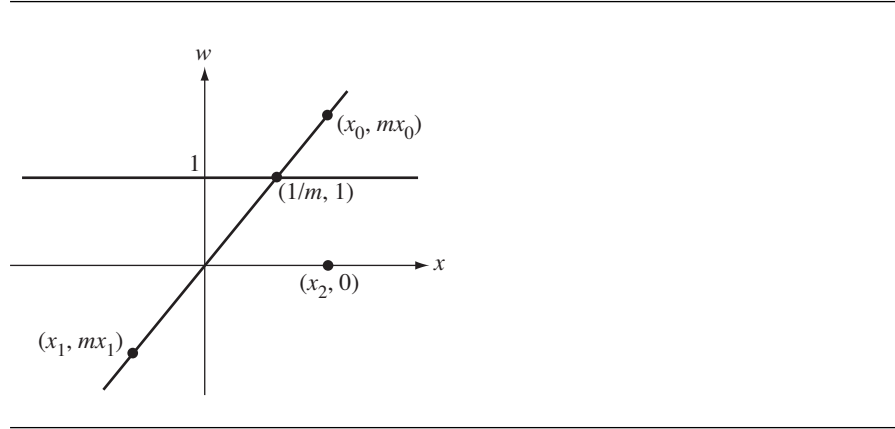


Figure 2.14 All homogeneous points along a line of slope m , excluding the origin, are equivalent to the homogeneous point $(1/m, 1)$. The points with a w -component of zero are vectors and are said to be equivalent to the *point at infinity*.

The division is not a matrix operation, because it involves a quantity dependent on the input $\mathbf{X}$. The use of the symbol $\sim$ in

$$\begin{bmatrix} \mathbf{Y} \\ 1 \end{bmatrix} \sim \begin{bmatrix} \mathbf{Y}' \\ w \end{bmatrix}$$

indicates that the 4-tuples are not equal but *equivalent* in the sense that the division by w does produce two equal 4-tuples. This equivalence is the basis for *projective geometry*.

To understand the equivalence in two dimensions, look at Figure 2.14. The figure shows a couple of homogeneous points, (x_0, mx_0) and (x_1, mx_1) , on the line of slope m . All homogeneous points on this line, excluding the origin, are equivalent to $(1/m, 1)$. The homogeneous point $(x_2, 0)$ corresponds to a vector since its w -component is zero. The division by zero cannot be performed, but the point is said to be equivalent to the *point at infinity*.

Now that we have the concept of equivalence of homogeneous points, notice that the matrix operations in the linear and affine transformations produce outputs whose w -component is 1. If we were to divide by w anyway, we would obtain the correct results for the transformations. This allows us finally to have a unifying representation for linear, affine, and perspective transformations—as homogeneous matrix operations. The most general form allows for inputs to have w -components that are not 1:

$$\begin{bmatrix} \mathbf{Y}' \\ w_1 \end{bmatrix} = \begin{bmatrix} M\mathbf{X}' + w_0\mathbf{B} \\ \mathbf{C}^T\mathbf{X}' + dw_0 \end{bmatrix} = \left[\begin{array}{c|c} M & \mathbf{B} \\ \hline \mathbf{C}^T & d \end{array} \right] \begin{bmatrix} \mathbf{X}' \\ w_0 \end{bmatrix} \quad (2.50)$$

When necessary, the equivalent point is computed by doing the perspective division. Linear transformations are characterized by $\mathbf{B} = \mathbf{0}$, $\mathbf{C} = \mathbf{0}$, $d = 1$, $w_0 = 1$, and $w_1 = 0$. Affine transformations are characterized by $\mathbf{C} = \mathbf{0}$, $d = 1$, $w_0 = 1$, and $w_1 = 0$. Perspective transformations occur when $\mathbf{C} \neq \mathbf{0}$ and, as long as $w_1 \neq 0$, you obtain the actual 3D projection point by doing the perspective divide.

Homogeneous transformations, specifically projective ones, are a major part of culling and clipping of triangles against the planes defining a view frustum; see Sections 2.3.5 and 2.4.3. They also occur in special effects such as planar projected shadows (Section 20.11), planar reflections (Section 20.10), projected textures (Section 20.12), and shadow maps (Section 20.13).

2.3 CAMERAS

Only a portion of the world is displayed at any one time. This region is called the *view volume*. Objects outside the view volume are not visible and therefore not drawn. The process of determining which objects are not visible is called *culling*. Objects that intersect the boundaries of the view volume are only partially visible. The visible portion of an object is determined by intersecting it with the view volume, a process called *clipping*.

The display of visible data is accomplished by projecting it onto a *view plane*. In this book I consider only perspective projection, as discussed in Sections 2.2.3 through 2.2.5. Orthogonal projection may also be used for viewing. In a graphics API, this amounts to choosing the parameters for a projection matrix.

2.3.1 THE PERSPECTIVE CAMERA MODEL

Our assumption is that the view volume is a bounded region in space, so the projected data lies in a bounded region in the view plane. A rectangular region in the view plane that contains the projected data is called a *viewport*. The viewport is what is drawn on the rectangular computer screen. The standard view volume used is called the *view frustum*. It is constructed by selecting an eye point and forming an infinite pyramid with four planar sides. Each plane contains the eye point and an edge of the viewport. The infinite pyramid is truncated by two additional planes called the *near plane* and the *far plane*. Figure 2.15 shows a view frustum.

The perspective projection is computed by intersecting a ray with the view plane. The ray has origin $\mathbf{E}$, the eye point, and passes through the world point $\mathbf{X}$. The intersection point is $\mathbf{Y}$. Equation (2.42) tells you how to construct $\mathbf{Y}$ from $\mathbf{X}$ as long as you know the eye point and the equation of the view plane, which I mention in the next paragraph. The combination of an eye point, a set of coordinate axes assigned to the eye point, a view plane, a viewport, and a view frustum is called a *camera model*.

The camera has a coordinate system associated with it. The *camera origin* is the eye point $\mathbf{E}$. The *camera view direction* is a unit-length vector $\mathbf{D}$ that is perpendicular

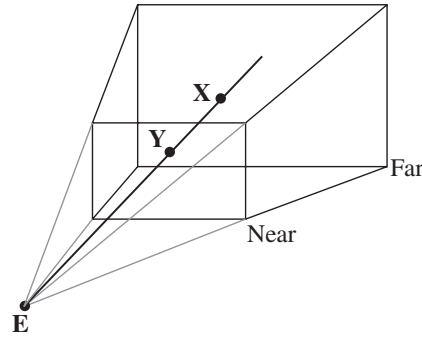


Figure 2.15 An eye point E and a view frustum. The point X in the view frustum is projected to the point Y in the viewport.

to the view plane. This direction vector is chosen to point away from the observer, so the eye point is considered to be on the negative side of the plane. If the view plane is at a distance $d_{\min}$ from the eye point, measured in the $\mathbf{D}$ direction, then the view plane normal to use in Equation (2.42) is $\mathbf{N} = -\mathbf{D}$ and the view plane point to use is $\mathbf{P} = \mathbf{E} + d_{\min}\mathbf{D}$. The *camera up vector* is the unit-length $\mathbf{U}$ vector chosen to be parallel to two opposing edges of the viewport. The *camera right vector* is the unit-length vector $\mathbf{R}$ chosen to be perpendicular to the camera direction and camera up vector with $\mathbf{R} = \mathbf{D} \times \mathbf{U}$. The coordinate system $\{\mathbf{E}; \mathbf{D}, \mathbf{U}, \mathbf{R}\}$ is a right-handed system.

Figure 2.16 shows the camera model, including the camera coordinate system and the view frustum. The six frustum planes are labeled with their names: near, far, left, right, bottom, top. The camera location E and the camera axis directions $\mathbf{D}$, $\mathbf{U}$, and $\mathbf{R}$ are shown. The view frustum has eight vertices. The near-plane vertices are $\mathbf{V}_{t\ell}$, $\mathbf{V}_{b\ell}$, $\mathbf{V}_{tr}$, and $\mathbf{V}_{br}$. Each subscript consists of two letters, the first letters of the frustum planes that share that vertex. The far-plane vertices have the name $\mathbf{W}$ and use the same subscript convention. The equations for the vertices are

$$\begin{aligned}
 \mathbf{V}_{b\ell} &= \mathbf{E} + d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\min}\mathbf{R} \\
 \mathbf{V}_{t\ell} &= \mathbf{E} + d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\min}\mathbf{R} \\
 \mathbf{V}_{br} &= \mathbf{E} + d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\max}\mathbf{R} \\
 \mathbf{V}_{tr} &= \mathbf{E} + d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\max}\mathbf{R}
 \end{aligned} \tag{2.51}$$

$$\mathbf{W}_{b\ell} = \mathbf{E} + \frac{d_{\max}}{d_{\min}} (d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\min}\mathbf{R})$$

$$\mathbf{W}_{t\ell} = \mathbf{E} + \frac{d_{\max}}{d_{\min}} (d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\min}\mathbf{R})$$

$$\mathbf{W}_{br} = \mathbf{E} + \frac{d_{\max}}{d_{\min}} (d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\max}\mathbf{R})$$

$$\mathbf{W}_{tr} = \mathbf{E} + \frac{d_{\max}}{d_{\min}} (d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\max}\mathbf{R})$$

The near plane is at a distance $d_{\min}$ from the camera location and the far plane is at a distance $d_{\max}$. These distances are the extreme values in the $\mathbf{D}$ direction. The extreme values in the $\mathbf{U}$ direction are $u_{\min}$ and $u_{\max}$. The extreme values in the $\mathbf{R}$ direction are $r_{\min}$ and $r_{\max}$.

The equations of the six view frustum planes are provided here in the form that is used for object culling. The near plane has inner-pointing, unit-length normal $\mathbf{D}$. A point on the plane is $\mathbf{E} + d_{\min}\mathbf{D}$. An equation of the near plane is

$$\mathbf{D} \cdot \mathbf{X} = \mathbf{D} \cdot (\mathbf{E} + d_{\min}\mathbf{D}) = \mathbf{D} \cdot \mathbf{E} + d_{\min} \quad (2.52)$$

The far plane has inner-pointing, unit-length normal $-\mathbf{D}$. A point on the plane is $\mathbf{E} + d_{\max}\mathbf{D}$. An equation of the far plane is

$$-\mathbf{D} \cdot \mathbf{X} = -\mathbf{D} \cdot (\mathbf{E} + d_{\max}\mathbf{D}) = -(\mathbf{D} \cdot \mathbf{E} + d_{\max}) \quad (2.53)$$

The left plane contains the three points $\mathbf{E}$, $\mathbf{V}_{t\ell}$, and $\mathbf{V}_{b\ell}$. A normal vector that points inside the frustum is

$$\begin{aligned} (\mathbf{V}_{b\ell} - \mathbf{E}) \times (\mathbf{V}_{t\ell} - \mathbf{E}) &= (d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\min}\mathbf{R}) \times (d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\min}\mathbf{R}) \\ &= (d_{\min}\mathbf{D} + r_{\min}\mathbf{R}) \times (u_{\max}\mathbf{U}) + (u_{\min}\mathbf{U}) \times (d_{\min}\mathbf{D} + r_{\min}\mathbf{R}) \\ &= (d_{\min}\mathbf{D} + r_{\min}\mathbf{R}) \times ((u_{\max} - u_{\min})\mathbf{U}) \\ &= (u_{\max} - u_{\min})(d_{\min}\mathbf{D} \times \mathbf{U} + r_{\min}\mathbf{R} \times \mathbf{U}) \\ &= (u_{\max} - u_{\min})(d_{\min}\mathbf{R} - r_{\min}\mathbf{D}) \end{aligned}$$

An inner-pointing, unit-length normal and the left plane are

$$\mathbf{N}_{\ell} = \frac{d_{\min}\mathbf{R} - r_{\min}\mathbf{D}}{\sqrt{d_{\min}^2 + r_{\min}^2}}, \quad \mathbf{N}_{\ell} \cdot (\mathbf{X} - \mathbf{E}) = 0 \quad (2.54)$$

An inner-pointing normal to the right plane is $(\mathbf{V}_{tr} - \mathbf{E}) \times (\mathbf{V}_{br} - \mathbf{E})$. A similar set of calculations as before will lead to an inner-pointing, unit-length normal and

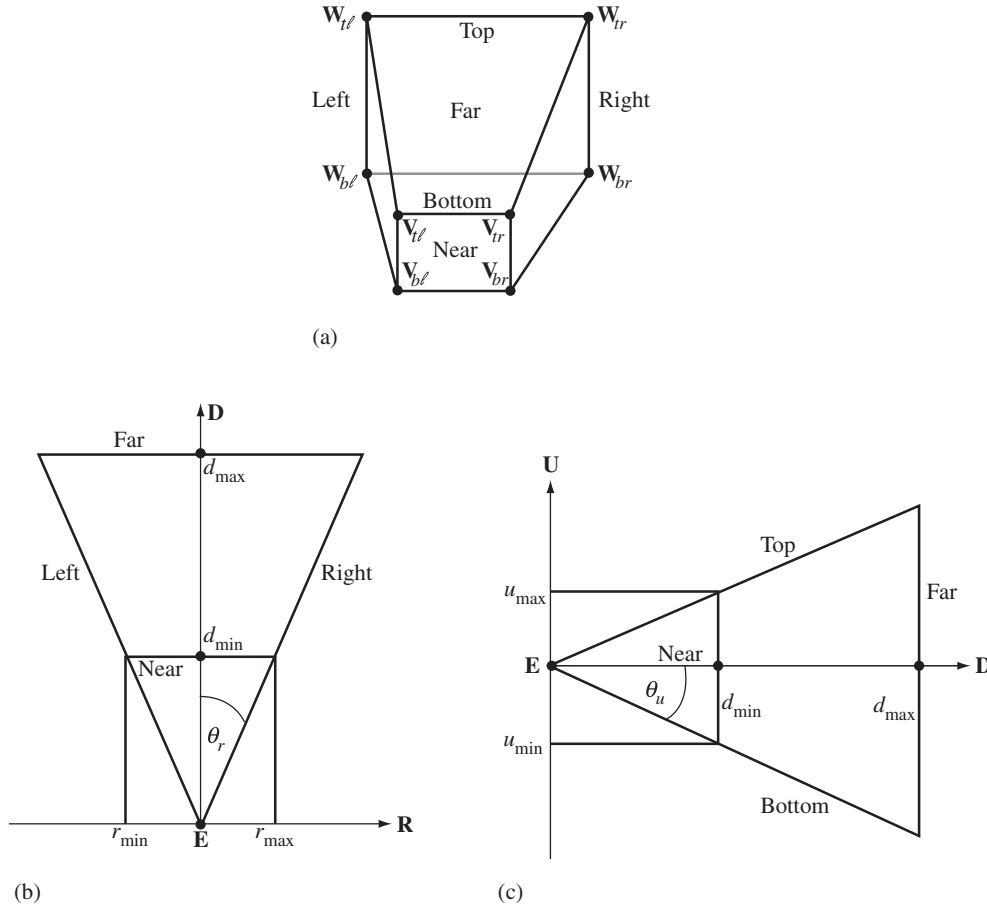


Figure 2.16 (a) A 3D drawing of the view frustum. The left, right, bottom, top, near, and far planes are labeled, as are the eight vertices of the frustum. (b) A 2D drawing of the view frustum as seen from the top side. (c) A 2D drawing of the view frustum as seen from the right side.

the right plane:

$$\mathbf{N}_r = \frac{-d_{\min}\mathbf{R} + r_{\max}\mathbf{D}}{\sqrt{d_{\min}^2 + r_{\max}^2}}, \quad \mathbf{N}_r \cdot (\mathbf{X} - \mathbf{E}) = 0 \quad (2.55)$$

Similarly, an inner-pointing, unit-length normal and the bottom plane are

$$\mathbf{N}_b = \frac{d_{\min} \mathbf{U} - u_{\min} \mathbf{D}}{\sqrt{d_{\min}^2 + u_{\min}^2}}, \quad \mathbf{N}_b \cdot (\mathbf{X} - \mathbf{E}) = 0 \quad (2.56)$$

An inner-pointing, unit-length normal and the top plane are

$$\mathbf{N}_t = \frac{-d_{\min} \mathbf{U} + u_{\max} \mathbf{D}}{\sqrt{d_{\min}^2 + u_{\max}^2}}, \quad \mathbf{N}_t \cdot (\mathbf{X} - \mathbf{E}) = 0 \quad (2.57)$$

It is common when choosing a camera model to have an *orthogonal view frustum*. The frustum is symmetric in that $u_{\min} = -u_{\max}$ and $r_{\min} = -r_{\max}$. The four independent frustum parameters are $d_{\min}$, $d_{\max}$, $u_{\max}$, and $r_{\max}$. An alternate way to specify the frustum is to use the *field of view* in the $\mathbf{U}$ direction and the *aspect ratio* for the viewport. In Figure 2.16 (c), the field of view is the angle $2\theta_u$. The aspect ratio is the width divided by height, in this case $\rho = r_{\max}/u_{\max}$. The frustum is completely determined by specifying $d_{\min}$, $d_{\max}$, θ_u , and ρ . The values for $u_{\max}$ and $r_{\max}$ are determined from

$$u_{\max} = d_{\min} \tan(\theta_u), \quad r_{\max} = \rho u_{\max} \quad (2.58)$$

The term *orthogonal* is used in this context to refer to the fact that the central axis of the frustum is orthogonal to the near face of the frustum. It does *not* refer to an orthographic projection.

Although every indication so far is that the projections of the points will be to the entire rectangular viewport of the view frustum, there are circumstances when you want to view a scene only in a subrectangle of the viewport. Using relative measurements, the full viewport is thought of as a unit square, as shown in Figure 2.17.

The full viewport has relative coordinates between 0 and 1. A smaller viewport is specified by choosing p_ℓ , p_r , p_b , and p_t so that $0 \leq p_\ell < p_r \leq 1$ and $0 \leq p_b < p_t \leq 1$. These relative coordinates will come into play when computing the actual pixel locations to draw in a window. The range of d values is $[d_{\min}, d_{\max}]$. A relative *depth range* is $[0, 1]$. The value 0 corresponds to $d_{\min}$ and the value 1 corresponds to $d_{\max}$. Some applications might want the depth range to be a subset $[p_n, p_f] \subseteq [0, 1]$.

In summary, you specify a perspective camera model by choosing an eye point $\mathbf{E}$; a right-handed orthonormal set of coordinate axis directions $\mathbf{D}$ (view direction), $\mathbf{U}$ (up direction), and $\mathbf{R}$ (right direction); the view frustum values $d_{\min}$ (near-plane distance from the eye point), $d_{\max}$ (far-plane distance from the eye point), $r_{\min}$ (minimum in right direction), $r_{\max}$ (maximum in right direction), $u_{\min}$ (minimum in up direction), and $u_{\max}$ (maximum in up direction); the viewport values p_ℓ (left), p_r (right), p_b (bottom), and p_t (top); and the depth range p_n (near) and p_f (far).

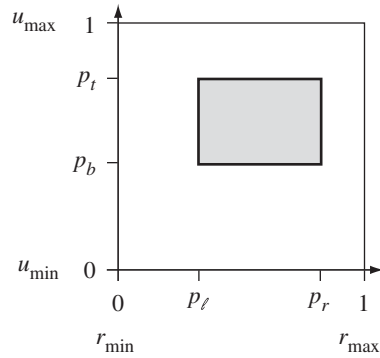


Figure 2.17 The full viewport of the view frustum is the full rectangle on the view plane. A smaller viewport is shown.

2.3.2 MODEL OR OBJECT SPACE

Three-dimensional modeling packages have their own specified coordinate systems for building polygonal models. I call the space in which the models are built *model space*. Others sometimes call this *object space*. I suppose if you are used to the art content being called models, you use model space, and if you are used to the content being called objects, you use object space.

2.3.3 WORLD SPACE

The coordinate system that is most prominent in a game is the *world coordinate system*, or *world space*. The choice is not important from a theoretical standpoint. From a practical standpoint, the choice might be related to constraints you place on the artists regarding the coordinate systems they use in their modeling packages. For example, if a modeling package has the convention that the positive y -axis is in the upward direction, then you might very well choose the world coordinates to use the positive y -axis for the upward direction. Most likely if you chose a world coordinate system for your previous project, you will choose the same one for the next project.

The main problem in dealing with both a world space and a model space is positioning, orienting, and possibly scaling the models so that they are correctly placed in the world. For example, Figure 2.18 (a) shows a tetrahedron built in a coordinate system provided by a modeling package.

The tetrahedron vertices in model space are $\mathbf{P}_0 = (0, 0, 0)$, $\mathbf{P}_1 = (1, 0, 0)$, $\mathbf{P}_2 = (0, 1, 0)$, and $\mathbf{P}_3 = (0, 0, 1)$. We want each tetrahedron vertex $\mathbf{P}_i$ to be located in world

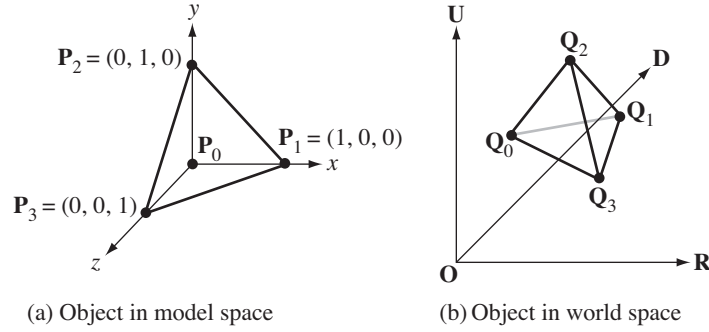


Figure 2.18 (a) A tetrahedron built in the model coordinate system. The origin is $(0, 0, 0)$ and the up direction is $(0, 1, 0)$. (b) The tetrahedron placed in the world coordinate system whose origin is O , whose view direction is D , whose up direction is $U = (0, 0, 1)$, and whose right direction is R .

space at the point $Q_i = O + d_i D + u_i U + r_i R$, $0 \leq i \leq 3$. This is accomplished by constructing an affine transformation that maps the point P_0 to the point Q_0 and that maps the vectors $P_i - P_0$ to the vectors $Q_i - Q_0$ for $1 \leq i \leq 3$. The transformation is

$$Q = Q_0 + M(P - P_0)$$

where $M(P_i - P_0) = Q_i - Q_0$. In algebraic terms, we need

$$M \begin{bmatrix} P_1 - P_0 & P_2 - P_0 & P_3 - P_0 \end{bmatrix} = \begin{bmatrix} Q_1 - Q_0 & Q_2 - Q_0 & Q_3 - Q_0 \end{bmatrix}$$

where the two block matrices have columns using the vectors as indicated. The matrix M is therefore

$$M = \begin{bmatrix} Q_1 - Q_0 & Q_2 - Q_0 & Q_3 - Q_0 \end{bmatrix} \begin{bmatrix} P_1 - P_0 & P_2 - P_0 & P_3 - P_0 \end{bmatrix}^{-1}$$

The matrix M is said to be the *model-to-world transformation* for the tetrahedron, sometimes called the *model transformation* and sometimes called the *world transformation*. Other transformations involved in converting model-space points to points in other spaces have names that indicate the range of the transformation—the set of outputs from the transformation. To be consistent with this terminology, I will use the term *world transformation*.

Given a 3×3 matrix M , which represents scaling, rotation, reflection, shearing, and other linear operations, given a 3×1 translation vector B , and given a 3×1 model-space point X_{model} , the corresponding 3×1 world-space point X_{world} is

generated by the homogeneous equation

$$\begin{bmatrix} \mathbf{X}_{\text{world}} \\ 1 \end{bmatrix} = \begin{bmatrix} M & \mathbf{B} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X}_{\text{model}} \\ 1 \end{bmatrix} = H_{\text{world}} \begin{bmatrix} \mathbf{X}_{\text{model}} \\ 1 \end{bmatrix} \quad (2.59)$$

The matrix H_{world} is the *world matrix* in homogeneous form. Naturally, as long as M is invertible, we can map world-space points to model-space points by

$$\begin{bmatrix} \mathbf{X}_{\text{model}} \\ 1 \end{bmatrix} = \begin{bmatrix} M^{-1} & -M^{-1}\mathbf{B} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X}_{\text{world}} \\ 1 \end{bmatrix} = H_{\text{world}}^{-1} \begin{bmatrix} \mathbf{X}_{\text{world}} \\ 1 \end{bmatrix} \quad (2.60)$$

where H_{world}^{-1} is the *inverse world matrix* in homogeneous form.

In the sample applications that ship with Wild Magic, the choice of the world space varies. Any objects that are loaded from disk are repositioned or reoriented as needed so that they are placed correctly in the world.

2.3.4 VIEW, CAMERA, OR EYE SPACE

So far we know about model space, the space where objects are created by the artists, and we know about world space, the space for the game environment itself. The objects are loaded into the game, but it is necessary to associate with them their world transformations. Model-space points are mapped to world-space points as needed.

A world-space point may also be located within the camera coordinate system. Once it is, the point is said to be in *view space* or *camera space* or *eye space* (all used by various people in the industry). The point must be represented as

$$\mathbf{X}_{\text{world}} = \mathbf{E} + d\mathbf{D} + u\mathbf{U} + r\mathbf{R}$$

where $\{\mathbf{E}; \mathbf{D}, \mathbf{U}, \mathbf{R}\}$ is the coordinate system for the camera. The coefficients are

$$d = \mathbf{D} \cdot (\mathbf{X}_{\text{world}} - \mathbf{E}), \quad u = \mathbf{U} \cdot (\mathbf{X}_{\text{world}} - \mathbf{E}), \quad r = \mathbf{R} \cdot (\mathbf{X}_{\text{world}} - \mathbf{E})$$

The eye point and camera directions are chosen to be consistent with your world coordinate system. In the beginning, there was nothing—except for Cartesian space. Your intent is to fill Cartesian space with your beautiful creations, and then place an observer in the world to admire them. Of course, this requires you to impose a world coordinate system. In many cases, you will have an idea of which direction you want to be the up direction. Two directions perpendicular to the up direction are chosen to complete your coordinate axes. The choice of origin is made. The world coordinates are of your choosing. How you position and orient the observer is a separate matter. Nothing prevents you from placing the observer on the ground standing on his head! However, the typical placement will be to have the observer's up direction align with the world's up direction. What the observer sees is determined by your camera model.

The coefficients of $\mathbf{X}$ in the camera coordinate system are stored in a 3×1 vector and referred to as the *view coordinates* for the world point,

$$\begin{aligned}\mathbf{X}_{\text{view}} &= \begin{bmatrix} r \\ u \\ d \end{bmatrix} = \begin{bmatrix} \mathbf{R} \cdot (\mathbf{X}_{\text{world}} - \mathbf{E}) \\ \mathbf{U} \cdot (\mathbf{X}_{\text{world}} - \mathbf{E}) \\ \mathbf{D} \cdot (\mathbf{X}_{\text{world}} - \mathbf{E}) \end{bmatrix} = \begin{bmatrix} \mathbf{R}^T \\ \mathbf{U}^T \\ \mathbf{D}^T \end{bmatrix} (\mathbf{X}_{\text{world}} - \mathbf{E}) \\ &= [\mathbf{R} \quad \mathbf{U} \quad \mathbf{D}]^T (\mathbf{X}_{\text{world}} - \mathbf{E})\end{aligned}$$

where the first equality defines $\mathbf{X}_{\text{view}}$. Please observe that I am listing the components in the order (r, u, d) , not in the natural order (d, u, r) that is associated with the coordinate system $\{\mathbf{E}; \mathbf{D}, \mathbf{U}, \mathbf{R}\}$! Effectively, (r, u, d) are the coordinates for the permuted coordinate system $\{\mathbf{E}; \mathbf{R}, \mathbf{U}, \mathbf{D}\}$, which happens to be left-handed. The Wild Magic software renderer implements the camera model in this way so that the last component of $\mathbf{X}_{\text{view}}$ is the view direction component. This choice was made to be consistent with the camera model of Direct3D, which is left-handed. OpenGL's camera coordinate system is internally stored as $\{\mathbf{E}; \mathbf{R}, \mathbf{U}, -\mathbf{D}\}$, which is right-handed. My initial attempt at dealing with this choice was to apply a sign change to the internal representation to produce $\mathbf{D}$. Having a consistent ordering is particularly important in vertex shader programs that transform and manipulate points and vectors in view space. My goal is to allow for the vertex shader programs to work with Wild Magic's software renderer, with the Direct3D renderer, and with the OpenGL renderer. The sample application for spherical environment mapping is a prototypical example where you manipulate view-space data.

Struggling with all the graphics APIs to make them consistent amounted to making programmatic adjustments to information obtained by API calls. For example, the camera coordinate system may be specified through Direct3D's utility functions `D3DXMATRIXLookAt*`. In OpenGL, the camera coordinate system may be specified through the utility function `gluLookAt`. For projections, Direct3D has utility functions `D3DXMatrixPerspective*` and `D3DXMatrixOrtho*`, whereas OpenGL has functions `glFrustum` and `glOrtho`. In the end, I tired of struggling and simply set the matrices directly—according to the coordinate system conventions I wanted, not the ones the graphics APIs want. The renderers were greatly simplified and a lot of code was factored into the base class for the renderers, a pleasant consequence. More details about this issue are found in Sections 2.8.3 and 2.8.4.

In homogeneous matrix form,

$$\begin{bmatrix} \mathbf{X}_{\text{view}} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{Q}^T & | & -\mathbf{Q}^T \mathbf{E} \\ \mathbf{0}^T & | & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X}_{\text{world}} \\ 1 \end{bmatrix} = H_{\text{view}} \begin{bmatrix} \mathbf{X}_{\text{world}} \\ 1 \end{bmatrix} \quad (2.61)$$

where $\mathbf{Q} = [\mathbf{R} \quad \mathbf{U} \quad \mathbf{D}]$ is the orthogonal matrix whose columns are the specified vectors. The homogenous matrix H_{view} in Equation (2.61) is referred to as the *view matrix*.

and maps points from world space to view space. Points may be mapped from view space to world space using the inverse,

$$\begin{bmatrix} \mathbf{X}_{\text{world}} \\ 1 \end{bmatrix} = \begin{bmatrix} Q & | & \mathbf{E} \\ \mathbf{0}^T & | & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X}_{\text{view}} \\ 1 \end{bmatrix} = H_{\text{view}}^{-1} \begin{bmatrix} \mathbf{X}_{\text{view}} \\ 1 \end{bmatrix} \quad (2.62)$$

Just a reminder: Section 2.8.3 goes into great detail on the view matrix representation for Wild Magic, OpenGL, and Direct3D. These details were essential in making a single vertex shader program work for *all* the graphics APIs. You should definitely read the details if you plan on using more than one graphics API.

2.3.5 CLIP, PROJECTION, OR HOMOGENEOUS SPACE

We are now ready to take our points in view coordinates and project them to obtain 2D coordinates for the screen. The process is factored into a few steps. The first step is to look more closely at the projection of Equation (2.42). We already looked at the projection in terms of camera coordinates in Section 2.2.4. The presentation here is in terms of homogeneous matrices so that you become comfortable with this approach rather than always relying on manipulating one component of a vector at a time.

The eye point $\mathbf{E}$ was chosen to be on the side of the projection plane (view plane) to which the normal vector $\mathbf{N}$ points. For our camera model, this direction is opposite to the view direction; namely, $\mathbf{N} = -\mathbf{D}$. A point on the view plane is $\mathbf{P} = \mathbf{E} + d_{\min}\mathbf{D}$. Using these choices and dividing the numerator and the denominator by -1 , Equation (2.42) becomes

$$\mathbf{Y} = \frac{(\mathbf{E}\mathbf{D}^T + d_{\min}I)(\mathbf{X} - \mathbf{E})}{\mathbf{D}^T(\mathbf{X} - \mathbf{E})}$$

The homogeneous form of this equation, which by convention does not include the perspective divide, and whose general form is Equation (2.49), is shown in the following equation:

$$\begin{bmatrix} \mathbf{Y}'_{\text{world}} \\ w_{\text{world}} \end{bmatrix} = \begin{bmatrix} \mathbf{E}\mathbf{D}^T + d_{\min}I & | & -(\mathbf{E}\mathbf{D}^T + d_{\min}I)\mathbf{E} \\ \mathbf{D}^T & | & -\mathbf{D}^T\mathbf{E} \end{bmatrix} \begin{bmatrix} \mathbf{X}_{\text{world}} \\ 1 \end{bmatrix}$$

Notice that I have subscripted the various terms to make it very clear that they are quantities in world coordinates. Since we already know how to map points from model space to world space, and then from world space to view space, it will be convenient to formulate the homogeneous equation so that its inputs are points in view space and its outputs are points in homogeneous view space, so to speak. We can convert the output from world space to view space using the view matrix of Equation

(2.61), replace the world-space input with the inverse view matrix of Equation (2.62) times the view-space input, and use $M = \mathbf{E}\mathbf{D}^T + d_{\min}\mathbf{I}$ to obtain

$$\begin{aligned}
 \begin{bmatrix} \mathbf{Y}'_{\text{view}} \\ w_{\text{view}} \end{bmatrix} &= \begin{bmatrix} \mathbf{Q}^T & -\mathbf{Q}^T\mathbf{E} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{Y}'_{\text{world}} \\ w_{\text{world}} \end{bmatrix} \\
 &= \begin{bmatrix} \mathbf{Q}^T & -\mathbf{Q}^T\mathbf{E} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{M} & -\mathbf{M}\mathbf{E} \\ \mathbf{D}^T & -\mathbf{D}^T\mathbf{E} \end{bmatrix} \begin{bmatrix} \mathbf{X}_{\text{world}} \\ 1 \end{bmatrix} \\
 &= \begin{bmatrix} \mathbf{Q}^T & -\mathbf{Q}^T\mathbf{E} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{M} & -\mathbf{M}\mathbf{E} \\ \mathbf{D}^T & -\mathbf{D}^T\mathbf{E} \end{bmatrix} \begin{bmatrix} \mathbf{Q} & \mathbf{E} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X}_{\text{view}} \\ 1 \end{bmatrix} \\
 &= \begin{bmatrix} d_{\min}\mathbf{Q}^T & -d_{\min}\mathbf{Q}^T\mathbf{E} \\ \mathbf{D}^T & -\mathbf{D}^T\mathbf{E} \end{bmatrix} \begin{bmatrix} \mathbf{Q} & \mathbf{E} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X}_{\text{view}} \\ 1 \end{bmatrix} \quad (2.63) \\
 &= \begin{bmatrix} d_{\min}\mathbf{I} & \mathbf{0} \\ \mathbf{D}^T\mathbf{Q} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{X}_{\text{view}} \\ 1 \end{bmatrix} \\
 &= \begin{bmatrix} d_{\min}\mathbf{X}_{\text{view}} \\ \mathbf{D}^T\mathbf{Q}\mathbf{X}_{\text{view}} \end{bmatrix} \\
 &= \begin{bmatrix} d_{\min}r \\ d_{\min}u \\ d_{\min}d \\ d \end{bmatrix}
 \end{aligned}$$

where you will recall that $\mathbf{X}_{\text{view}} = (r, u, d)$. The perspective divide produces the actual projection,

$$\mathbf{Y}_{\text{proj}} = \frac{\mathbf{Y}'_{\text{view}}}{w_{\text{view}}} = \begin{bmatrix} \frac{d_{\min}r}{d} \\ \frac{d_{\min}u}{d} \\ d_{\min} \end{bmatrix} \quad (2.64)$$

The last component of $\mathbf{Y}_{\text{proj}}$ makes sense because the view plane is $d_{\min}$ units of distance from the eye point and we designed the projection to be onto the view plane.

The axis of the view frustum is the ray that contains both the origin and the center point of the viewport. This ray is parameterized by d in view coordinates as

$$\left(\frac{(r_{\min} + r_{\max})d}{2d_{\min}}, \frac{(u_{\min} + u_{\max})d}{2d_{\min}}, d \right), \quad d_{\min} \leq d \leq d_{\max}$$

It is convenient to transform the (possibly) skewed view frustum into an orthogonal frustum with viewport $[-1, 1]^2$. We accomplish this by removing the skew, then scaling the result:

$$\begin{aligned} r' &= \frac{2}{r_{\max} - r_{\min}} \left(d_{\min} r - \frac{r_{\min} + r_{\max}}{2} d \right), \\ u' &= \frac{2}{u_{\max} - u_{\min}} \left(d_{\min} u - \frac{u_{\min} + u_{\max}}{2} d \right) \end{aligned} \quad (2.65)$$

To keep consistent with the primed notation r' and u' , define

$$w' = d \quad (2.66)$$

The view frustum is now delimited by $|r'| \leq w'$, $|u'| \leq w'$, and $d_{\min} \leq w' \leq d_{\max}$. The projection is $(r'/w', u'/w')$, so $|r'/w'| \leq 1$ and $|u'/w'| \leq 1$.

It is also convenient to transform the d -values in $[d_{\min}, d_{\max}]$ so that the new range is $[0, 1]$. This is somewhat tricky because the transformation should be consistent with the perspective projection. The affine transformation $d' = (d - d_{\min}) / (d_{\max} - d_{\min})$ is not the correct one to use. Equation (2.43) saves the day. The d -values in $[d_{\min}, d_{\max}]$ can be written as

$$d = (1 - s)d_{\min} + sd_{\max}$$

for $s \in [0, 1]$. We can solve this for $s = (d - d_{\min}) / (d_{\max} - d_{\min})$ and use Equation (2.43) with $w_0 = d_{\min}$, the minimum w' -value, and $w_1 = d_{\max}$, the maximum w' -value, to obtain

$$\bar{s} = \frac{w_1 s}{w_0 + (w_1 - w_0)s} = \frac{d_{\max}}{d_{\max} - d_{\min}} \left(1 - \frac{d_{\min}}{d} \right)$$

Observe that $\bar{s} \in [0, 1]$. This value plays the role of a *normalized depth* in rendering. The equation for $\bar{s}$ already has the perspective division. Before division, we can define

$$d' = \frac{d_{\max}}{d_{\max} - d_{\min}} (d - d_{\min}) \quad (2.67)$$

so that $\bar{s} = d'/w'$.

Equations (2.65) through (2.67) may be combined into a homogeneous matrix transformation that maps $(r, u, d, 1)$ to (r', u', d', w') :

$$\begin{aligned}
\mathbf{X}_{\text{clip}} &= \begin{bmatrix} r' \\ u' \\ d' \\ w' \end{bmatrix} \\
&= \left[\begin{array}{ccc|c} \frac{2d_{\min}}{r_{\max}-r_{\min}} & 0 & \frac{-(r_{\max}+r_{\min})}{r_{\max}-r_{\min}} & 0 \\ 0 & \frac{2d_{\min}}{u_{\max}-u_{\min}} & \frac{-(u_{\max}+u_{\min})}{u_{\max}-u_{\min}} & 0 \\ 0 & 0 & \frac{d_{\max}}{d_{\max}-d_{\min}} & \frac{-d_{\max}d_{\min}}{d_{\max}-d_{\min}} \\ \hline 0 & 0 & 1 & 0 \end{array} \right] \begin{bmatrix} r \\ u \\ d \\ 1 \end{bmatrix} \quad (2.68) \\
&= H_{\text{proj}} \begin{bmatrix} \mathbf{X}_{\text{view}} \\ 1 \end{bmatrix}
\end{aligned}$$

This equation defines two quantities, the homogeneous *projection matrix* H_{proj} and the homogeneous point $\mathbf{X}_{\text{clip}}$, which is a point said to be in *clip space* and its components are referred to as *clip coordinates*.

Clip coordinates are used both for culling back-facing triangles and for clipping triangles against the view frustum. Although you could do these calculations in world space or in view space, the number of calculations is fewer in clip space. Moreover, the access to the graphics pipeline provided via vertex shaders essentially requires you to compute points in clip coordinates, which are then returned to the graphics driver for rasterization.

All that said, you might have looked at Equation (2.68) and concluded that it looks neither like OpenGL's projection matrix nor like Direct3D's projection matrix. I will explicitly compare these in Sections 2.8.3 and 2.8.4. Suffice it to say that my OpenGL and Direct3D renderers have been implemented to use the exact same view and projection matrices, thereby ignoring the defaults that occur when you go through utility functions provided by the graphics APIs.

Just as I have provided the inverses for the world matrix and the view matrix, the inverse of the projection matrix is

$$H_{\text{proj}}^{-1} = \begin{bmatrix} \frac{r_{\max}-r_{\min}}{2d_{\min}} & 0 & 0 & \frac{r_{\max}+r_{\min}}{2d_{\min}} \\ 0 & \frac{u_{\max}-u_{\min}}{2d_{\min}} & 0 & \frac{u_{\max}+u_{\min}}{2d_{\min}} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{d_{\max}-d_{\min}}{d_{\max}d_{\min}} & \frac{1}{d_{\min}} \end{bmatrix} \quad (2.69)$$

2.3.6 WINDOW SPACE

The clip-space point (r', u', d', w') has the properties that $|r'| \leq w'$, $|u'| \leq w'$, $0 \leq d' \leq d_{\max}$, and $d_{\min} \leq w' \leq d_{\max}$. We finally perform the perspective division to obtain

$$\mathbf{X}_{\text{ndc}} = \begin{bmatrix} r'' \\ u'' \\ d'' \\ 1 \end{bmatrix} = \begin{bmatrix} r'/w' \\ u'/w' \\ d'/w' \\ w'/w' \end{bmatrix} \quad (2.70)$$

where $r'' \in [-1, 1]$, $u'' \in [-1, 1]$, and $d'' \in [0, 1]$. The 3-tuples (r'', u'', d'') are said to be *normalized device coordinates* (NDCs). The term *normalized* was intended to refer to the components of the 3-tuples being somehow in intervals $[0, 1]$ or $[-1, 1]$. The normalization, however, is not normal across APIs. Wild Magic and Direct3D use $d'' \in [0, 1]$. OpenGL has a default projection matrix that leads to a projected value $d'' \in [-1, 1]$. This is yet another API convention you need to be aware of; see Section 2.8.4 for more details. But as I have mentioned repeatedly, my implementations of the renderers all use the same projection matrix, so in fact my OpenGL renderer has $d'' \in [0, 1]$.

The goal now is to map (r'', u'') to a pixel of the window created by your application. One important detail is that (r'', u'') are right-handed coordinates with respect to the viewport on the view plane. The r'' -values increase as you move to the right within the viewport and the u'' -values increase as you move up within the viewport. The window pixels have coordinates (x, y) that are left-handed. The x -values increase as you move to the right in the window and the y -values increase as you move down the window. The conversion from (r'', u'') to (x, y) requires a reflection in u'' to switch handedness. If the window has width W pixels and height H pixels, then $0 \leq x < W$, $0 \leq y < H$, and a mapping is $x = W(1 + r'')/2$ and $y = H(1 - u'')/2$. The computations are real-valued, but in software the values are truncated to the nearest integer and then clamped to be within the valid pixel domain to produce the indices into video memory for the screen. This mapping takes clip-space points to the full viewport on the view plane. As mentioned in Section 2.3.1, you might want the drawing of objects to occur in a subrectangle of the viewport. The camera model includes parameters p_ℓ , p_r , p_b , and p_t with $0 \leq p_\ell < p_r \leq 1$ and $0 \leq p_b < p_t \leq 1$. The mapping from $(r'', u'') \in [-1, 1]^2$ to the subrectangle is

$$\begin{aligned} x &= \left(\frac{1 - r''}{2} \right) p_\ell W + \left(\frac{1 + r''}{2} \right) p_r W = \frac{W}{2} [(p_r + p_\ell) + (p_r - p_\ell)r''] \\ y &= H - \left[\left(\frac{1 - u''}{2} \right) p_b H + \left(\frac{1 + u''}{2} \right) p_t H \right] \\ &= \frac{H}{2} [(2 - p_t - p_b) + (p_b - p_t)u''] \end{aligned} \quad (2.71)$$

The slightly more complicated conversion for u'' has to do with the switch from right-handed to left-handed coordinates.

The depth values $d'' \in [0, 1]$ can also be mapped to a depth range that is a subset of $[0, 1]$. Section 2.3.1 introduced the depth range interval $[p_n, p_f] \subseteq [0, 1]$. The new depth values for this range are

$$\delta = (p_f - p_n) d'' + p_n \quad (2.72)$$

Equations (2.71) and (2.72) may be combined into a single 4-tuple, which I will call the *window coordinates* of the corresponding clip-space point:

$$\begin{aligned} \begin{bmatrix} \mathbf{X}_{\text{window}} \\ 1 \end{bmatrix} &= \begin{bmatrix} \frac{W(p_r - p_\ell)}{2} & 0 & 0 & \frac{W(p_r + p_\ell)}{2} \\ 0 & \frac{H(p_b - p_t)}{2} & 0 & \frac{H(2 - p_t - p_b)}{2} \\ 0 & 0 & p_f - p_n & p_n \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X}_{\text{ndc}} \\ 1 \end{bmatrix} \\ &= H_{\text{window}} \begin{bmatrix} \mathbf{X}_{\text{ndc}} \\ 1 \end{bmatrix} \end{aligned} \quad (2.73)$$

The CD-ROM accompanying this book contains a software renderer that implements all the transformations described in this section. The vertex shader unit takes model-space points and produces clip-space points. The rasterizer clips the points and generates the pixels that are covered by a triangle via interpolation. Each interpolated clip-space point is mapped to a window-space point to produce the pixel location and depth. The pixel shader unit processes each such pixel.

EXERCISE 2.10

The window matrix of Equation (2.73) was developed using the mapping of $r'' \in [-1, 1]$ to $x \in [p_\ell W, p_r W]$ and $u'' \in [-1, 1]$ to $y \in [p_b H, p_t H]$, with a reflection when computing the y -value. This choice was made to be consistent with OpenGL, according to the documentation describing this mapping. The DirectX documentation [Cor] does not mention the precise details of the mapping. When the full viewport is used ($p_\ell = 0$, $p_r = 1$, $p_b = 0$, $p_t = 1$), notice that $r'' = 1$ is mapped to $x = W$ and $u'' = -1$ is mapped to $y = H$, but actual pixel coordinates must satisfy $x \leq W - 1$ and $y \leq H - 1$, so clamping will always occur at these extremes. What differences in visual behavior would you expect if you were to use a different mapping?

One alternative is to map r'' to $x \in [p_\ell W, p_r W - 1]$ and u'' to $y \in [p_b H, p_t H - 1]$, with a reflection. What is the window matrix for this transformation?

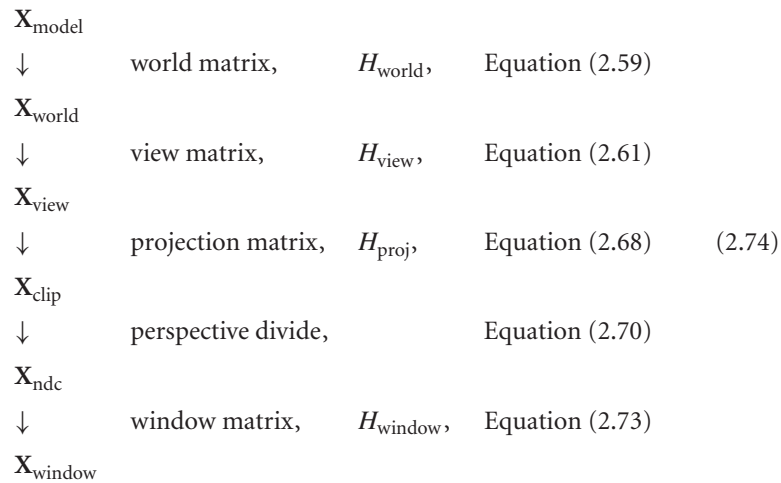
Another alternative is to map r'' to $x \in [p_\ell(W - 1), p_r(W - 1)]$ and u'' to $y \in [p_b(H - 1), p_t(H - 1)]$, with a reflection. What is the window matrix for this transformation?

Try experimenting with these in the software renderer contained on this book's companion CD-ROM. For each suggested alternative, also modify the OpenGL and Direct3D rendering code (functions `OnViewportChange`) and see how the visual behavior changes. ■

EXERCISE 2.11 What is the inverse matrix for the window matrix of Equation (2.73)? ■

2.3.7 PUTTING THEM ALL TOGETHER

The application of transformations from model space to window space is referred to as the *geometric pipeline*. The following diagram shows all the steps, including references to the equations that define the transformations.



A software renderer implements the entire geometric pipeline. The companion CD-ROM has such a renderer to illustrate the concepts discussed in this book. A hardware-accelerated renderer implements the pipeline and allows you, through a graphics API, to specify the matrices in the pipeline, either directly or indirectly.

EXAMPLE 2.4 A triangle is created in a model space with points labeled (x, y, z) . The model-space vertices are $\mathbf{V}_0 = (0, 0, 0)$, $\mathbf{V}_1 = (1, 0, 0)$, and $\mathbf{V}_2 = (0, 0, 1)$. Figure 2.19 shows a rendering of the triangle in model space. The world space is chosen with origin $(0, 0, 0)$ and with an up vector in the direction of the positive z -axis. The model triangle is to be rotated and translated so that the world-space vertices are $\mathbf{W}_0 = (1, 1, 1)$, $\mathbf{W}_1 = (1, 2, 1)$, and $\mathbf{W}_2 = (1, 1, 2)$. Figure 2.20 shows a rendering of the triangle in world space. The world matrix is

$$H_{\text{world}} = \left[\begin{array}{ccc|c} 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 \end{array} \right]$$

and transforms $(\mathbf{V}_i, 1)$ to $(\mathbf{W}_i, 1)$ for all i .

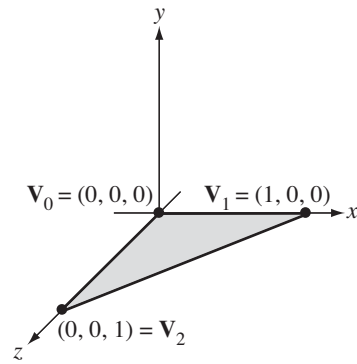


Figure 2.19

A model triangle to be sent through the geometric pipeline.

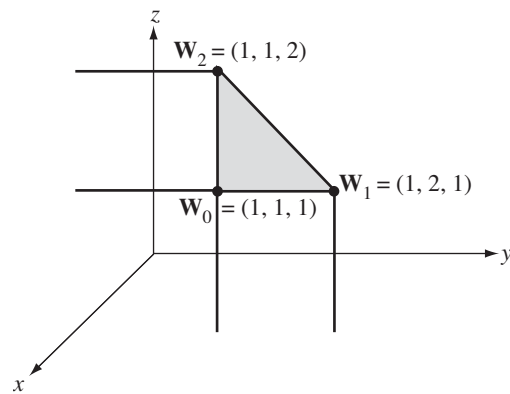


Figure 2.20

The world triangle corresponding to the model triangle of Figure 2.19.

The screen is chosen to have a width of 640 pixels and a height of 480 pixels. The camera is positioned in the world with eye point at $E = (5/2, 3, 7/2)$, with view direction $D = (-1, -1, -1)/\sqrt{3}$, and up direction $U = (-1, -1, 2)/\sqrt{6}$. The right direction is $R = D \times U = (-1, 1, 0)/\sqrt{2}$. The view matrix is

$$H_{\text{view}} = \left[\begin{array}{ccc|c} \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & \frac{-1}{2\sqrt{2}} \\ \frac{-1}{\sqrt{6}} & \frac{-1}{\sqrt{6}} & \frac{2}{\sqrt{6}} & \frac{-3}{2\sqrt{6}} \\ \frac{-1}{\sqrt{3}} & \frac{-1}{\sqrt{3}} & \frac{-1}{\sqrt{3}} & \frac{9}{\sqrt{3}} \\ \hline 0 & 0 & 0 & 1 \end{array} \right]$$

An orthogonal frustum will be used to render the triangle. The frustum near and far parameters are chosen to be $d_{\min} = 1$ and $d_{\max} = 10$, respectively. The vertical field of view is chosen to be $2\theta_u = \pi/3$ and the aspect ratio is $\rho = 4/3 = 640/480$. Equation (2.58) is used to compute $u_{\max} = d_{\min} \tan(\theta_u) = 1/\sqrt{3}$ and $r_{\max} = \rho u_{\max} = 4/(3\sqrt{3})$. By symmetry, $u_{\min} = -u_{\max}$ and $r_{\min} = -r_{\max}$. The projection matrix is

$$H_{\text{proj}} = \left[\begin{array}{ccc|c} \frac{3\sqrt{3}}{4} & 0 & 0 & 0 \\ 0 & \sqrt{3} & 0 & 0 \\ 0 & 0 & \frac{10}{9} & \frac{-10}{9} \\ \hline 0 & 0 & 1 & 0 \end{array} \right]$$

We will use the full viewport, so $p_\ell = p_b = 0$ and $p_r = p_t = 1$. Also, we will use the full depth range, so $p_n = 0$ and $p_f = 1$. The screen matrix is

$$H_{\text{screen}} = \left[\begin{array}{ccc|c} \frac{639}{2} & 0 & 0 & \frac{639}{2} \\ 0 & \frac{-479}{2} & 0 & \frac{479}{2} \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array} \right]$$

All the matrices are ready to use, so let us transform the model-space vertices and see where they land on the screen. To make Table 2.2 typesetting friendly, I will write the 4-tuples in the form $(a, b, c; d)$, using a semicolon to separate the last component from the first three. Computing the screen-space coordinates, the final points (x, y) and normalized depths $\delta \in [0, 1]$ are

$$(x_0, y_0; \delta_0) = (277.139884, 312.831599; 0.790360)$$

$$(x_1, y_1; \delta_1) = (370.332138, 386.163198; 0.726210)$$

$$(x_2, y_2; \delta_2) = (268.667861, 210.167360; 0.726210)$$

Table 2.2 Results of the transformations applied during the geometric pipeline.

<i>Space</i>	<i>Vertex 0</i>	<i>Vertex 1</i>	<i>Vertex 2</i>
Model	(0, 0, 0; 1)	(1, 0, 0; 1)	(0, 0, 1; 1)
World	(1, 1, 1; 1)	(1, 2, 1; 1)	(1, 1, 2; 1)
View	$\left(\frac{-1}{2\sqrt{2}}, \frac{-3}{2\sqrt{6}}, \frac{6}{\sqrt{3}}; 1\right)$	$\left(\frac{1}{2\sqrt{2}}, \frac{-5}{2\sqrt{6}}, \frac{5}{\sqrt{3}}; 1\right)$	$\left(\frac{-1}{2\sqrt{2}}, \frac{1}{2\sqrt{6}}, \frac{5}{\sqrt{3}}; 1\right)$
Clip	$\left(\frac{-3\sqrt{3}}{8\sqrt{2}}, \frac{-3}{2\sqrt{2}}, \frac{20}{3\sqrt{3}} - \frac{10}{9}, \frac{6}{\sqrt{3}}\right)$	$\left(\frac{3\sqrt{3}}{8\sqrt{2}}, \frac{-5}{2\sqrt{2}}, \frac{50}{9\sqrt{3}} - \frac{10}{9}, \frac{5}{\sqrt{3}}\right)$	$\left(\frac{-3\sqrt{3}}{8\sqrt{2}}, \frac{1}{2\sqrt{2}}, \frac{50}{9\sqrt{3}} - \frac{10}{9}, \frac{5}{\sqrt{3}}\right)$
NDC	$\left(\frac{-3}{16\sqrt{2}}, \frac{-\sqrt{3}}{4\sqrt{2}}, \frac{10}{9} - \frac{5\sqrt{3}}{27}; 1\right)$	$\left(\frac{9}{40\sqrt{2}}, \frac{-\sqrt{3}}{2\sqrt{2}}, \frac{10}{9} - \frac{2\sqrt{3}}{9}; 1\right)$	$\left(\frac{-9}{40\sqrt{2}}, \frac{\sqrt{3}}{10\sqrt{2}}, \frac{10}{9} - \frac{2\sqrt{3}}{9}; 1\right)$
Screen	$\left(\frac{639}{2} - \frac{1917}{32\sqrt{2}}, \frac{479}{2} + \frac{479\sqrt{3}}{8\sqrt{2}}, \frac{10}{9} - \frac{5\sqrt{3}}{27}; 1\right)$	$\left(\frac{639}{2} + \frac{5751}{80\sqrt{2}}, \frac{479}{2} + \frac{479\sqrt{3}}{4\sqrt{2}}, \frac{10}{9} - \frac{2\sqrt{3}}{9}; 1\right)$	$\left(\frac{639}{2} - \frac{5751}{80\sqrt{2}}, \frac{479}{2} - \frac{479\sqrt{3}}{20\sqrt{2}}, \frac{10}{9} - \frac{2\sqrt{3}}{9}; 1\right)$

The x and y values are rounded to the nearest integer, so the actual pixel locations for the projected vertices are (277, 313), (370, 386), and (269, 210). Figure 2.21 shows the final image drawn by the Wild Magic software renderer to a 640×480 window. The coordinate axes were drawn as three separate polylines. The 640×480 image was reduced in size, with averaging, to a 320×240 image. The border around the window and the axis labels were added via a paint program. ■

Naturally, the geometric pipeline is part of the rendering system. The application code that led to Figure 2.21 created the model-space triangle, the world matrix, and a simple scene, and it did some basic setup for rendering. The application header file is

```
#ifndef GEOMETRICPIPELINE_H
#define GEOMETRICPIPELINE_H

#include "Wm4WindowApplication3.h"
using namespace Wm4;

class GeometricPipeline : public WindowApplication3
{
    WM4_DECLARE_INITIALIZE;
```

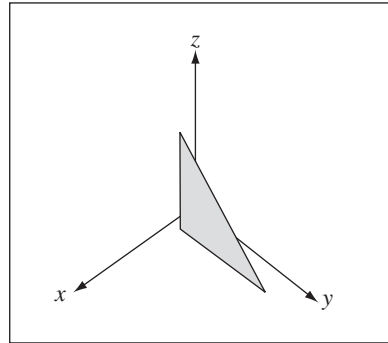


Figure 2.21 A software-rendered image of the triangle.

```

public:
    GeometricPipeline ();

    virtual bool OnInitialize ();
    virtual void OnTerminate ();
    virtual void OnIdle ();

protected:
    void CreateScene ();

    NodePtr m_spkScene;
    TriMeshPtr m_spkTriangle;
    PolylinePtr m_spkAxes;
    Culler m_kCuller;
};

WM4_REGISTER_INITIALIZE(GeometricPipeline);

#endif

```

The application source code is

```

#include "GeometricPipeline.h"

WM4_WINDOW_APPLICATION(GeometricPipeline);

```

```

//-----
GeometricPipeline::GeometricPipeline ()
:
    WindowApplication3("GeometricPipeline",0,0,640,480,ColorRGBA::WHITE)
{
}
//-----
bool GeometricPipeline::OnInitialize ()
{
    if (!WindowApplication3::OnInitialize())
    {
        return false;
    }

    // Create the camera model.
    m_spkCamera->SetFrustum(60.0f,4.0f/3.0f,1.0f,10.0f);
    Vector3f kCLoc(2.5f,3.0f,3.5f);
    Vector3f kCDir(-1.0f,-1.0f,-1.0f);
    kCDir.Normalize();
    Vector3f kCUUp(-1.0f,-1.0f,2.0f);
    kCUUp.Normalize();
    Vector3f kCRight = kCDir.Cross(kCUUp);
    m_spkCamera->SetFrame(kCLoc,kCDir,kCUUp,kCRight);

    CreateScene();

    // The initial update of objects.
    m_spkScene->UpdateGS();
    m_spkScene->UpdateRS();

    // The initial culling of the scene.
    m_kCuller.SetCamera(m_spkCamera);
    m_kCuller.ComputeVisibleSet(m_spkScene);

    InitializeCameraMotion(0.1f,0.01f);
    InitializeObjectMotion(m_spkScene);
    return true;
}
//-----
void GeometricPipeline::OnTerminate ()
{
    m_spkScene = 0;
    m_spkTriangle = 0;
}

```

```

        m_spkAxes = 0;
        WindowApplication3::OnTerminate();
    }
    //-----
    void GeometricPipeline::OnIdle ()
    {
        MeasureTime();

        if (MoveCamera())
        {
            m_kCuller.ComputeVisibleSet(m_spkScene);
        }

        if (MoveObject())
        {
            m_spkScene->UpdateGS();
            m_kCuller.ComputeVisibleSet(m_spkScene);
        }

        m_pkRenderer->ClearBuffers();
        if (m_pkRenderer->BeginScene())
        {
            m_pkRenderer->DrawScene(m_kCuller.GetVisibleSet());
            DrawFrameRate(8,GetHeight()-8,ColorRGBA::WHITE);
            m_pkRenderer->EndScene();
        }
        m_pkRenderer->DisplayBackBuffer();

        UpdateFrameCount();
    }
    //-----
    void GeometricPipeline::CreateScene ()
    {
        // Create the model-space triangle.
        Attributes kAttr;
        kAttr.SetPChannels(3);
        VertexBuffer* pkVBuffer = WM4_NEW VertexBuffer(kAttr,3);
        pkVBuffer->Position3(0) = Vector3f(0.0f,0.0f,0.0f);
        pkVBuffer->Position3(1) = Vector3f(1.0f,0.0f,0.0f);
        pkVBuffer->Position3(2) = Vector3f(0.0f,0.0f,1.0f);

        IndexBuffer* pkIBuffer = WM4_NEW IndexBuffer(3);
        int* aiIndex = pkIBuffer->GetData();
    }

```

```

aiIndex[0] = 0;
aiIndex[1] = 1;
aiIndex[2] = 2;

m_spkTriangle = WM4_NEW TriMesh(pkVBuffer,pkIBuffer);

// Set the world matrix.
m_spkTriangle->Local.SetTranslate(Vector3f(1.0f,1.0f,1.0f));
m_spkTriangle->Local.SetRotate(Matrix3f(Vector3f::UNIT_Z,Mathf::HALF_PI));

// Attach a material to the triangle.
MaterialState* pkMS = WM4_NEW MaterialState;
pkMS->Diffuse = ColorRGB(0.5f,0.5f,0.5f);
m_spkTriangle->AttachGlobalState(pkMS);
m_spkTriangle->AttachEffect(WM4_NEW MaterialEffect);

// Create the coordinate axes.
pkVBuffer = WM4_NEW VertexBuffer(kAttr,6);
pkVBuffer->Position3(0) = Vector3f::ZERO;
pkVBuffer->Position3(1) = 2.0f*Vector3f::UNIT_X;
pkVBuffer->Position3(2) = Vector3f::ZERO;
pkVBuffer->Position3(3) = 2.0f*Vector3f::UNIT_Y;
pkVBuffer->Position3(4) = Vector3f::ZERO;
pkVBuffer->Position3(5) = 2.0f*Vector3f::UNIT_Z;

m_spkAxes = WM4_NEW Polyline(pkVBuffer,false,false);

// Attach a material to the axes.
pkMS = WM4_NEW MaterialState;
pkMS->Diffuse = ColorRGB::BLACK;
m_spkAxes->AttachGlobalState(pkMS);
m_spkAxes->AttachEffect(WM4_NEW MaterialEffect);

m_spkScene = WM4_NEW Node;
m_spkScene->AttachChild(m_spkTriangle);
m_spkScene->AttachChild(m_spkAxes);
}
//-----

```

In Wild Magic, the application layer is agnostic of renderer type. The code works for the OpenGL renderer, for the Direct3D renderer, and for the Wild Magic software renderer.

- EXERCISE 2.12** Repeat the calculations in Example 2.4, but using a camera positioned at $E = (4, 4, 4)$ and with a far-plane distance of $d_{\max} = 4$. Also repeat the calculations with the original settings, except place the camera at $E = (1, -1, 3/2)$. How is the rendering of the triangle in this case different from the rendering in Figure 2.21? ■
- EXERCISE 2.13** Suppose you want your application to support selecting a window pixel with the left button of the mouse. When the selected pixel is part of a rendered 3D object, compute the world-space coordinates for the 3D object point that was rendered to the selected pixel. Add this code to the `GeometricPipeline` application whose source code was listed previously. Write text to the upper-left corner of the screen that displays the (x, y) value you selected with the mouse and the corresponding world-space coordinates of the object drawn to that pixel. ■

2.4 CULLING AND CLIPPING

Culling and clipping of objects reduces the amount of data sent to the rasterizer for drawing. Culling refers to eliminating portions of an object, possibly the entire object, that are not visible to the eye point. For an object represented by a triangle mesh, the typical culling operations amount to determining which triangles are outside the view frustum and which triangles are facing away from the eye point. Clipping refers to computing the intersection of an object with the view frustum, and with additional planes provided by the application such as in a portal system (see Section 6.3), so that only the visible portion of the object is sent to the rasterizer. For an object represented by a triangle mesh, the typical clipping operations amount to splitting triangles by the various view frustum planes and retaining only those triangles inside the frustum.

2.4.1 OBJECT CULLING

Object culling involves deciding whether or not an object as a whole is contained in the view frustum. If an object is not in the frustum, there is no point in consuming CPU or GPU cycles to process the object for the rasterizer. Typically, the application maintains a bounding volume for each object. The idea is to have an inexpensive test for nonintersection between bounding volume and view frustum that can lead to quick rejection of an object for further processing. If the bounding volume of an object does intersect the view frustum, then the entire object is processed further even if that object does not lie entirely inside the frustum. It is also possible that the bounding volume and view frustum intersect, but the object is completely outside the frustum.

A test to determine if the bounding volume and view frustum intersect can be an expensive operation. Such a test is said to be an *exact culling test*. An *inexact culling test* is designed to be faster, reporting nonintersections in *most* cases, but is conservative in that it might report an intersection when there is none. The idea is that the total

time for culling and drawing is, hopefully, less than the total time if you were to use exact culling. Specifically, what you hope to be the common situation is

$$\begin{aligned}\text{Cost}(\text{inexact\_culling}) &< \text{Cost}(\text{exact\_culling}) \\ \text{Cost}(\text{inexact\_drawing}) &> \text{Cost}(\text{exact\_drawing}) \\ \text{Cost}(\text{inexact\_culling}) + \text{Cost}(\text{inexact\_drawing}) &< \text{Cost}(\text{exact\_culling}) + \\ &\quad \text{Cost}(\text{exact\_drawing})\end{aligned}$$

The only way you can test this hypothesis is by experimenting within your own applications and graphics framework. If you find that over the lifetime of your application's execution the total time of culling and drawing is smaller when using exact culling, then you should certainly use exact culling. Some exact culling tests are described in Section 15.7.

The standard approach to inexact culling against the view frustum is to compare the object's bounding volume against the view frustum planes, one at a time. Figure 2.22 illustrates the various possibilities for culling by testing a plane at a time. The situation shown in Figure 2.22 (a) occurs whether you use exact culling or inexact culling of bounding volumes. The problem is simply that the bounding volume is an approximation of the region that the object occupies; there will always be situations when the bounding volume intersects the frustum but the object does not. The situation shown in Figure 2.22 (c) is what makes the plane-by-plane culling inexact. The bounding volume is not outside any frustum plane, but it is outside the entire view frustum.

2.4.2 BACK-FACE CULLING

Object culling is an attempt to eliminate the entire object from being processed by the renderer. If an object is not culled based on its bounding volume, then the renderer has the opportunity to reduce the amount of data it must draw. The next level of culling is called *back-face culling*. The triangles are oriented so that their normal vectors point outside the object whose surface they comprise. If the triangle is oriented away from the eye point, then that triangle is not visible and need not be drawn by the renderer. For a perspective projection, the test for a back-facing triangle is to determine if the eye point is on the negative side of the plane of the triangle (the triangle is a “back face” of the object to be rendered). If $\mathbf{E}$ is the world eye point and if the plane of the triangle is $\mathbf{N} \cdot \mathbf{X} = d$, then the triangle is back facing if $\mathbf{N} \cdot \mathbf{E} < d$. Figure 2.23 shows the front view of an object. The front-facing triangles are drawn with solid lines. The back-facing triangles are indicated with dotted lines (although they would not be drawn at all by the renderer).

The vertex data that is sent to the graphics driver stores only vertex positions, not triangle normals. This means the renderer must compute the normal vector for each triangle to use in the back-face test. Mathematically, it does not matter in which

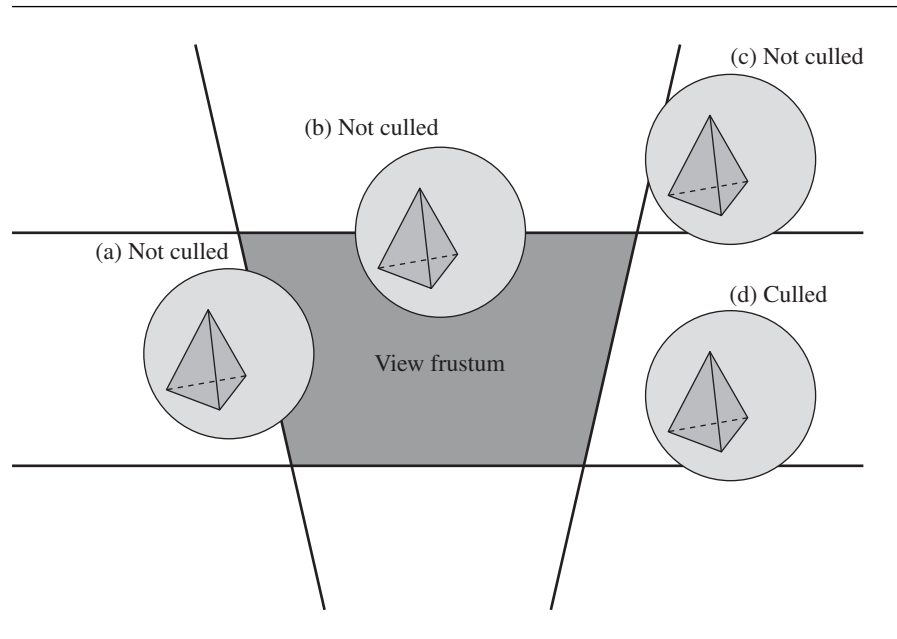


Figure 2.22 Attempts to cull objects, whose bounding volumes are spheres, a frustum plane at a time. In (a), (b), and (c), the bounding volumes are not outside any of the frustum planes, so an attempt will be made to draw those objects. In (a), the bounding volume is not outside any of the frustum planes, so an attempt is made to draw the object. The object is outside the frustum even though its bounding volume is not. The renderer processes the object and determines that no part of it will be drawn on the screen. In (b), part of the object is inside the frustum, so the renderer will draw that portion. In (c), the object and its bounding volume are outside the frustum, but because the bounding volume was not outside at least one of the frustum planes, the object is sent to the renderer and it is determined that no part of it will be drawn on the screen. In (d), the bounding volume is outside the right plane of the frustum, so the object is outside and no attempt is made to draw it.

coordinate system you do the back-face culling. However, vertex shader programs require you to transform the vertex positions from model-space coordinates to clip-space coordinates for the purpose of clipping, so it is natural to do the back-face culling in these same coordinates. The transformation of the triangle vertices from model space to view space produces points $\mathbf{V}_i = (r_i, u_i, d_i, 1)$ for $0 \leq i \leq 2$. A triangle normal vector is

$$\mathbf{N} = (\mathbf{V}_1 - \mathbf{V}_0) \times (\mathbf{V}_2 - \mathbf{V}_0)$$

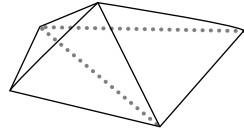


Figure 2.23 Object with front-facing and back-facing triangles indicated.

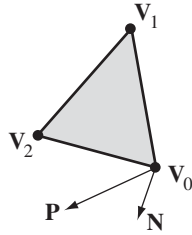


Figure 2.24 A triangle that is front facing to the observer. Because the camera coordinate system is left-handed, the sign test for the dot product of vectors is the opposite of what you are used to.

The eye point in view coordinates is $\mathbf{P} = (0, 0, 0, 1)$. Figure 2.24 shows the situation when the triangle is deemed visible to the observer. The vector $\mathbf{P} - \mathbf{V}_0 = (-r_0, -u_0, -d_0, 0)$ must form an acute angle with the normal vector $\mathbf{N}$. The test for the triangle to be front facing is

$$0 < (\mathbf{P} - \mathbf{V}_0) \cdot \mathbf{N} = \det \begin{bmatrix} -r_0 & r_1 - r_0 & r_2 - r_0 \\ -u_0 & u_1 - u_0 & u_2 - u_0 \\ -d_0 & d_1 - d_0 & d_2 - d_0 \end{bmatrix}$$

Define the homogeneous matrix

$$M = \begin{bmatrix} r_0 & r_1 & r_2 & 0 \\ u_0 & u_1 & u_2 & 0 \\ d_0 & d_1 & d_2 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

The first three columns of the matrix are the triangle vertices and the last column of the matrix is the eye point, all listed in view coordinates. The determinant of the matrix is computed as follows, using a cofactor expansion in the last column.

$$\begin{aligned}
\det(M) &= -\det \begin{bmatrix} r_0 & r_1 & r_2 \\ u_0 & u_1 & u_2 \\ d_0 & d_1 & d_2 \end{bmatrix} && \text{Cofactor expansion is by last column.} \\
&= -\det \begin{bmatrix} r_0 & r_1 - r_0 & r_2 - r_0 \\ u_0 & u_1 - u_0 & u_2 - u_0 \\ d_0 & d_1 - d_0 & d_2 - d_0 \end{bmatrix} && \text{Subtracting columns preserves} \\
&&& \text{determinants.} \\
&= \det \begin{bmatrix} -r_0 & r_1 - r_0 & r_2 - r_0 \\ -u_0 & u_1 - u_0 & u_2 - u_0 \\ -d_0 & d_1 - d_0 & d_2 - d_0 \end{bmatrix} && \text{Changing column sign} \\
&&& \text{reverses determinant sign.} \\
&= (\mathbf{P} - \mathbf{V}_0) \cdot \mathbf{N}
\end{aligned}$$

Thus, the triangle is visible when $\det(M) > 0$.

Multiplying M by the projection matrix of Equation (2.68), we have

$$H_{\text{proj}}M = \begin{bmatrix} r'_0 & r'_1 & r'_2 & 0 \\ u'_0 & u'_1 & u'_2 & 0 \\ d'_0 & d'_1 & d'_2 & -\frac{d_{\max}d_{\min}}{d_{\max}-d_{\min}} \\ w'_0 & w'_1 & w'_2 & 0 \end{bmatrix}$$

Using a cofactor expansion on the last column, we may compute the determinant of this matrix:

$$\det(H_{\text{proj}}M) = -\frac{d_{\max}d_{\min}}{d_{\max}-d_{\min}} \begin{bmatrix} r'_0 & r'_1 & r'_2 \\ u'_0 & u'_1 & u'_2 \\ w'_0 & w'_1 & w'_2 \end{bmatrix}$$

A front-facing triangle occurs when $\det(M) > 0$, so equivalently it occurs when $\det(H_{\text{proj}}M) = \det(H_{\text{proj}}) \det(M) < 0$. That is, the triangle is front facing when

$$\det \begin{bmatrix} r'_0 & r'_1 & r'_2 \\ u'_0 & u'_1 & u'_2 \\ w'_0 & w'_1 & w'_2 \end{bmatrix} > 0$$

This expression is what the Wild Magic software renderer implements, and is found in the file `Wm4SoftDrawElements.cpp`, function `SoftRenderer::DrawTriMesh`.

2.4.3 CLIPPING TO THE VIEW FRUSTUM

Clipping is the process by which the front-facing triangles of an object in the world are intersected with the view frustum planes. A triangle either is completely inside the frustum (no clipping necessary), is completely outside the frustum (triangle is

culled), or intersects at least one frustum plane (needs clipping). In the last case the portion of the triangle that lies on the frustum side of the clipping plane must be calculated. That portion is either a triangle itself or a quadrilateral.

Plane-at-a-Time Clipping

One possibility for a simple clipping algorithm is to clip the triangle against a frustum plane. If the portion inside the frustum is a triangle, process that triangle against the next frustum plane. If the portion inside the frustum is a quadrilateral, split it into two triangles and process both against the next frustum plane. After all clipping planes are processed, the renderer has a list of triangles that are completely inside the view frustum. The pseudocode for this process is shown next.

```

set<Triangle> input, output;
input.Insert(initialTriangle);
for each frustum plane do
{
    for each triangle in input do
    {
        set<Triangle> inside = Split(triangle,plane);
        if (inside.Quantity() == 2)
        {
            output.Insert(inside.Element[0]);
            output.Insert(inside.Element[1]);
        }
        else if (inside.Quantity() == 1)
        {
            output.Insert(inside.Element[0]);
        }
        else
        {
            // Inside is empty, triangle is culled.
        }
        input.Remove(triangle);
    }
    input = output;
}

for each triangle in output do
{
    // Draw the triangle.
}

```

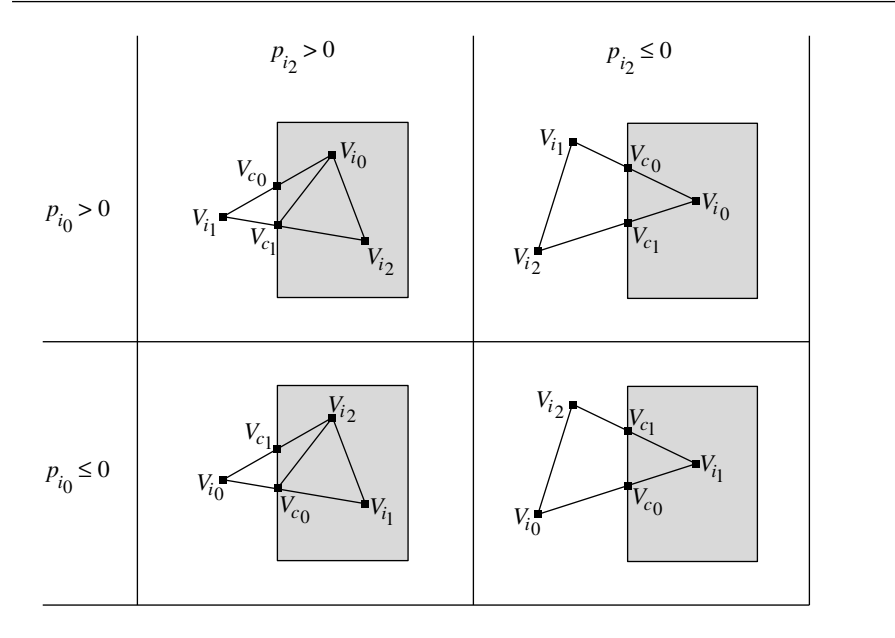


Figure 2.25 Four configurations for triangle splitting. Only the triangles in the shaded region are important, so the quadrilaterals outside are not split. The subscript c indicates clip vertices.

The splitting of a triangle by a frustum plane is accomplished by computing the intersection of the triangle edges with the plane. The three vertices of the triangle are tested for inclusion in the frustum. If the frustum plane is $\mathbf{N} \cdot \mathbf{X} = d$ and if the vertices of the triangle are $\mathbf{V}_i$ for $0 \leq i \leq 2$, then the edge with endpoints $\mathbf{V}_{i_0}$ and $\mathbf{V}_{i_1}$ intersects the plane if $p_{i_0}p_{i_1} < 0$, where $p_i = \mathbf{N} \cdot \mathbf{V}_i - d$ for $0 \leq i \leq 2$. This simply states that one vertex is on the positive side of the plane and one vertex is on the negative side of the plane. The point of intersection, called a *clip vertex*, is

$$\mathbf{V}_{\text{clip}} = \mathbf{V}_{i_0} + \frac{p_{i_0}}{p_{i_0} - p_{i_1}} (\mathbf{V}_{i_1} - \mathbf{V}_{i_0}) \quad (2.75)$$

Figure 2.25 illustrates the possible configurations for clipping of a triangle against a plane. The vertices $\mathbf{V}_{i_0}$, $\mathbf{V}_{i_1}$, and $\mathbf{V}_{i_2}$ are assumed to be in counterclockwise order.

The pseudocode for clipping a single triangle against a plane is given next. After splitting, the new triangles have vertices that are in counterclockwise order.

```
void ClipConfiguration (pi0,pi1,pi2,Vi0,Vi1,Vi2)
{
    // assert: pi0*pi1 < 0
```

```

Vc0 = Vi0+(pi0/(pi0-pi1))*(Vi1-Vi0);
if (pi0 > 0)
{
    if (pi2 > 0) // Figure 2.25, top left
    {
        Vc1 = Vi1+(pi1/(pi1-pi2))*(Vi2-Vi1);
        add triangle <Vc0,Vc1,Vi0> to triangle list;
        add triangle <Vc1,Vi2,Vi0> to triangle list;
    }
    else // Figure 2.25, top right
    {
        Vc1 = Vi0+(pi0/(pi0-pi2))*(Vi2-Vi0);
        add triangle <Vc0,Vc1,Vi0> to triangle list;
    }
}
else
{
    if (pi2 > 0) // Figure 2.25, bottom left
    {
        Vc1 = Vi0+(pi0/(pi0-pi2))*(Vi2-Vi0);
        add triangle <Vc0,Vi1,Vi2> to triangle list;
        add triangle <Vc0,Vi2,Vc1> to triangle list;
    }
    else // Figure 2.25, bottom right
    {
        Vc1 = Vi1+(pi1/(pi1-pi2))*(Vi2-Vi1);
        add triangle <Vc0,Vi1,Vc1> to triangle list;
    }
}
}

void ClipTriangle ()
{
    remove triangle <V0,V1,V2> from triangle list;

    p0 = Dot(N,V0)-d;
    p1 = Dot(N,V1)-d;
    p2 = Dot(N,V2)-d;

    if (p0*p1 < 0)
    {
        // Triangle needs splitting along edge <V0,V1>.
        ClipConfiguration(p0,p1,p2,V0,V1,V2);
    }
    else if (p0*p2 < 0)

```



```

{
    // Triangle needs splitting along edge <V0,V2>.
    ClipConfiguration(p2,p0,p1,V2,V0,V1);
}
else if (p1*p2 < 0)
{
    // Triangle needs splitting along edge <V1,V2>.
    ClipConfiguration(p1,p2,p0,V1,V2,V0);
}
else if (p0 > 0 || p1 > 0 || p2 > 0)
{
    // Triangle is completely inside frustum.
    add triangle <V0,V1,V2> to triangle list;
}
}

```

To avoid copying vertices, the triangle representation can store pointers to vertices in a vertex pool, adding clip vertices as needed.

Polygon-of-Intersection Clipping

The plane-at-a-time clipping algorithm keeps track of a set of triangles that must be clipped against frustum planes. Processing only triangles leads to simple data structures and algorithms. The drawback is that the number of triangles can be larger than is really necessary.

An alternate method for clipping computes the convex polygon of intersection of the triangle with the frustum. After clipping, a triangle fan is generated for the polygon and these triangles are drawn. The number of triangles in this approach is smaller than or equal to the number produced by the plane-at-a-time clipping algorithm. An illustration of this is provided by the sequence of images shown in Figures 2.26 through 2.30. For the sake of simplicity, the example is shown in two dimensions with the frustum drawn as a rectangle. Figure 2.26 shows a triangle intersecting a frustum. The convex polygon of intersection has seven vertices. The triangle fan is drawn, indicating that the renderer will draw five triangles.

Let us clip the triangle against the four frustum planes one at a time. Figure 2.27 shows the triangle clipped against the bottom frustum plane. Two clip vertices are generated. The portion of the triangle on the frustum side of the bottom plane is a quadrilateral, so it is split into two triangles T_1 and T_2 .

Figure 2.28 shows the triangles clipped against the top frustum plane. The triangle T_1 is clipped, generates two clip vertices, and is split into two triangles, T_3 and T_4 . The triangle T_2 is clipped, generates two clip vertices, and is split into two triangles, T_5 and T_6 .

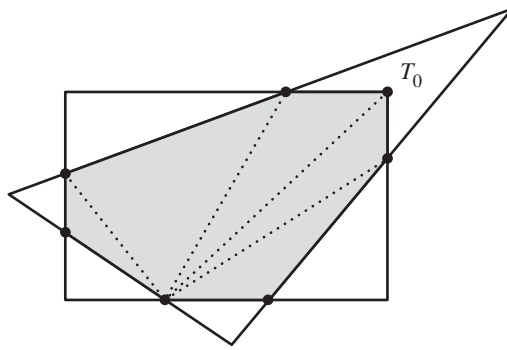


Figure 2.26 A triangle T_0 intersecting a frustum in multiple faces. The convex polygon of intersection has seven vertices and is represented by a triangle fan with five triangles.

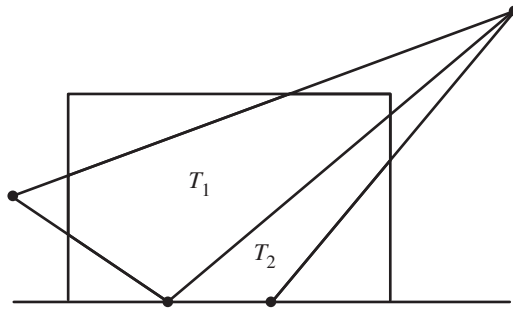


Figure 2.27 The triangle is clipped against the bottom frustum plane.

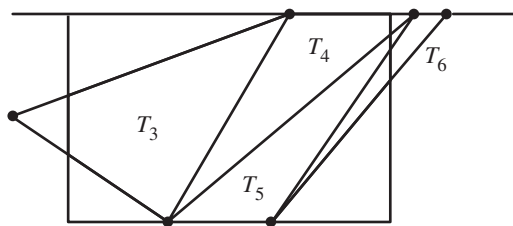


Figure 2.28 The triangles are clipped against the top frustum plane.

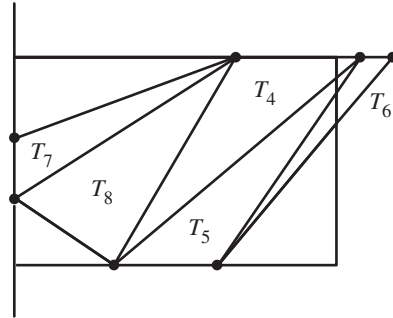


Figure 2.29 The triangles are clipped against the left frustum plane.

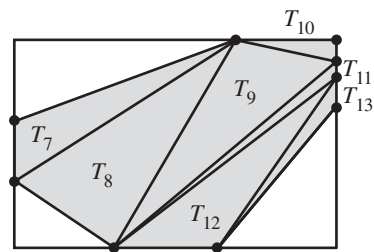


Figure 2.30 The triangles are clipped against the right frustum plane.

Figure 2.29 shows the triangles clipped against the left frustum plane. In this case, only triangle T_3 intersects the left frustum plane. It generates two clip vertices and the quadrilateral inside the frustum is split into two triangles, T_7 and T_8 .

Finally, Figure 2.30 shows the triangles clipped against the right frustum plane. Triangle T_4 is clipped and split into triangles T_9 and T_{10} . Triangle T_5 is clipped and split into triangles T_{11} and T_{12} . Triangle T_6 is clipped, producing a single triangle T_{13} . The end result is a collection of nine vertices and seven triangles in contrast to the polygon-of-intersection clipping algorithm, which produced seven vertices and five triangles.

At first glance, the polygon-of-intersection clipping algorithm is attractive because it tends to generate fewer triangles than the plane-at-a-time clipping algorithm. However, the example here is slightly misleading because the triangle is very large compared to the frustum size. In a realistic application, the observer is positioned so that triangles are generally small compared to the frustum size, so you would expect a

triangle to be clipped by one frustum plane (triangle intersects a face of the frustum), by two frustum planes (triangle intersects near an edge of the frustum), or three frustum planes (triangle intersects near a corner of the frustum). In these cases, either clipping method should perform equally well.

EXERCISE 2.14 The Wild Magic software renderer implements the polygon-of-intersection clipping algorithm. Modify the renderer to use the plane-at-a-time clipping algorithm. Devise an experiment to test the performance of the two clipping algorithms and compare the results. ■

2.5 RASTERIZING

Rasterization is the process of taking a geometric entity in window space and selecting those pixels to be drawn that correspond to the entity. The standard objects that most engines rasterize are line segments and triangles, but rasterization of circles and ellipses is also discussed here. You might have a situation where you want to rasterize such objects to a texture and then use the texture for one of your 3D objects. The constructions contained in this section all assume integer arithmetic since the main goal is to rasterize as fast as possible. Floating-point arithmetic tends to be more expensive than integer arithmetic.

EXERCISE 2.15 This is a large project. The Wild Magic software renderer uses floating-point arithmetic for its rasterization; that is, the renderer is not optimized for speed (it was designed to illustrate concepts). If you feel adventuresome, reimplement the rasterizing code to use integer arithmetic. This code is found in files `Wm4SoftDrawElements.cpp` and `Wm4SoftEdgeBuffers.cpp`. ■

2.5.1 LINE SEGMENTS

Given two screen points (x_0, y_0) and (x_1, y_1) , a line segment must be drawn that connects them. Since the pixels form a discrete set, decisions must be made about which pixels to draw in order to obtain the “best” line segment, which Figure 2.31 illustrates. If $x_1 = x_0$ (vertical segment) or $y_1 = y_0$ (horizontal segment), it is clear which pixels to draw. And if $|x_1 - x_0| = |y_1 - y_0|$, the segment is diagonal and it is clear which pixels to draw. But for the other cases, it is not immediately apparent which pixels to draw.

The algorithm should depend on the magnitude of the slope. If the magnitude is larger than 1, each row that the segment intersects should have a pixel drawn. If the magnitude is smaller than 1, each column that the segment intersects should have a pixel drawn. Figure 2.32 illustrates the cases. The two blocks of pixels in (a) illustrate the possibilities for drawing pixels for a line with a slope whose magnitude is larger than 1. The case in (a) draws one pixel per column. The case in (b) draws one pixel

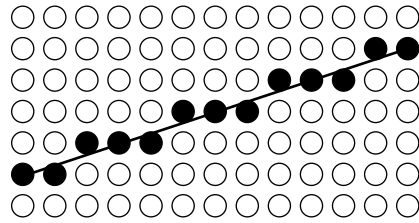


Figure 2.31 Pixels that form the best line segment between two points.

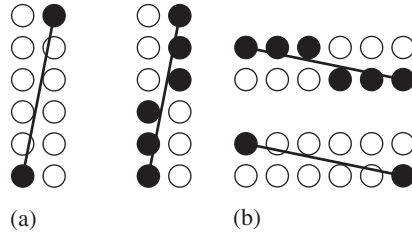


Figure 2.32 Pixel selection based on slope.

per row, the correct decision. The two blocks of pixels in (b) illustrate the possibilities for drawing pixels for a line with a slope whose magnitude is less than 1. The bottom case draws one pixel per row. The top case draws one pixel per column, the correct decision.

The process of pixel selection, called Bresenham's algorithm [Bre65], uses an integer decision variable that is updated for each increment in the appropriate input variable. The sign of the decision variable is used to select the correct pixel to draw at each step. Define $dx = x_1 - x_0$ and $dy = y_1 - y_0$. For the sake of argument, assume that $dx > 0$ and $dy \neq 0$. The decision variable is d_i , and its value is determined by the pixel (x_i, y_i) that was drawn at the previous step. Figure 2.33 shows two values s_i and t_i , the fractional lengths of the line segment connecting two vertical pixels. The value of s_i is determined by $s_i = (y_0 - y_i) + (dy/dx)(x_i + 1 - x_0)$ and $s_i + t_i = 1$. The decision variable is $d_i = dx(s_i - t_i)$. From the figure it can be seen that

- If $d_i \geq 0$, then the line is closer to the pixel at $(x_i + 1, y_i + 1)$, so draw that pixel.
- If $d_i < 0$, then the line is closer to the pixel at $(x_i + 1, y_i)$, so draw that pixel.

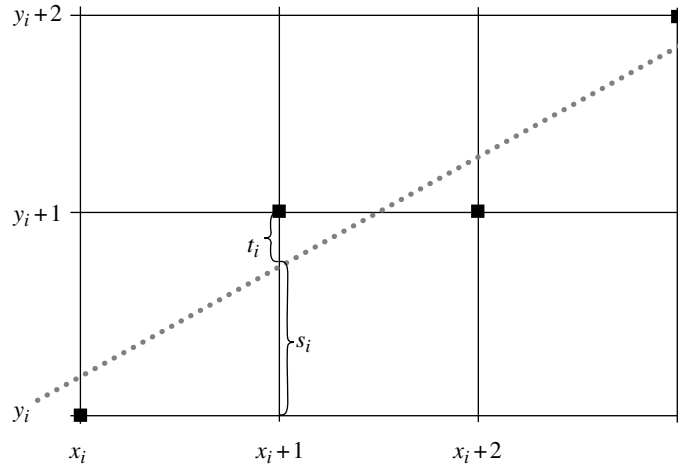


Figure 2.33 Deciding which line pixel to draw next.

Now consider

$$\begin{aligned}
 d_{i+1} - d_i &= dx(s_{i+1} - t_{i+1}) - dx(s_i - t_i) \\
 &= 2dx(s_{i+1} - s_i) \\
 &= 2dy(x_{i+1} - x_i) - 2dx(y_{i+1} - y_i).
 \end{aligned}$$

The initial decision value is $d_0 = 2dy - dx$. Figure 2.33 indicates that the slope has a magnitude less than 1, so x is incremented in the drawing, $x_{i+1} = x_i + 1$. The decision equation is therefore

$$d_{i+1} = d_i + 2dy - 2dx(y_{i+1} - y_i)$$

and the rules for setting the next pixel are

- If $d_i \geq 0$, then $y_{i+1} = y_i + 1$ and the next decision value is $d_{i+1} = d_i + 2(dy - dx)$.
- If $d_i < 0$, then $y_{i+1} = y_i$ and the next decision value is $d_{i+1} = d_i + 2dy$.

A concise implementation is given next. The special cases of horizontal, vertical, and diagonal lines can be factored out if desired.

```

void DrawLine (int x0, int y0, int x1, int y1)
{
    // starting point of line
    int x = x0, y = y0;

    // direction of line
    int dx = x1 - x0, dy = y1 - y0;

    // Increment or decrement depending on direction of line.
    int sx, sy;
    if (dx > 0)
    {
        sx = 1;
    }
    else if (dx < 0)
    {
        sx = -1;
        dx = -dx;
    }
    else
    {
        sx = 0;
    }

    if (dy > 0)
    {
        sy = 1;
    }
    else if (dy < 0)
    {
        sy = -1;
        dy = -dy;
    }
    else
    {
        sy = 0;
    }

    int ax = 2*dx, ay = 2*dy;

    if (dy <= dx)
    {
        // single step in x-direction

```

```

for (int decy = ay-dx; /**/; x += sx, decy += ay)
{
    DrawPixel(x,y);

    // Take Bresenham step.
    if (x == x1)
    {
        break;
    }
    if (decy >= 0)
    {
        decy -= ax;
        y += sy;
    }
}
}
else
{
    // single step in y-direction
    for (int decx = ax-dy; /**/; y += sy, decx += ax)
    {
        DrawPixel(x,y);

        // Take Bresenham step.
        if (y == y1)
        {
            break;
        }
        if (decx >= 0)
        {
            decx -= ay;
            x += sx;
        }
    }
}
}

```

In the line-drawing algorithm, the calls `DrawLine(x0,y0,x1,y1)` and `DrawLine(x1,y1,x0,y0)` can produce different sets of drawn pixels. It is possible to avoid this by using a variation called the *midpoint line algorithm*; the midpoint $(x_m, y_m) = ((x_0 + x_1)/2, (y_0 + y_1)/2)$ is computed, then two line segments are drawn, `DrawLine(xm,ym,x0,y0)` and `DrawLine(xm,ym, x1,y1)`. This is particularly useful if a line segment is drawn twice, something that happens when rasterizing triangles that share

an edge. If the original line drawer is used for the shared edge, but the line is drawn the second time with the endpoints swapped, gaps (undrawn pixels) can occur because the two sets of drawn pixels cause an effect called *cracking*. Another way to avoid cracking is to always draw the line starting with the vertex of the minimum y -value. This guarantees that the shared edge is drawn in the same order each time.

2.5.2 CIRCLES

The Bresenham line-drawing algorithm has a counterpart for drawing circles using only integer arithmetic. Let the circle be $x^2 + y^2 = r^2$, where r is a positive integer. The algorithm will draw one-eighth of the circle for $y \geq x \geq 0$. The remaining parts are drawn by symmetry.

Let (x_0, y_0) be the last drawn pixel. Let $\mathbf{A} = (x_0 + 1, y_0)$ and $\mathbf{B} = (x_0 + 1, y_0 - 1)$. A decision must be made about which of the two points should be drawn next. Figure 2.34 illustrates the various possibilities. The selected pixel will be the one closest to the circle measured in terms of radial distance from the origin. The squared distance will be calculated to avoid square roots.

Define $D(x, y) = x^2 + y^2$; then $D(\mathbf{A}) = (x_0 + 1)^2 + y_0^2$ and $D(\mathbf{B}) = (x_0 + 1)^2 + (y_0 - 1)^2$. Define $f(x, y) = D(x, y) - r^2$. If $f(\mathbf{P}) > 0$, then $\mathbf{P}$ is outside the circle. If $f(\mathbf{P}) < 0$, then $\mathbf{P}$ is inside the circle. Finally, if $f(\mathbf{P}) = 0$, then $\mathbf{P}$ is on the circle. The rules for setting pixels are

1. If $|f(\mathbf{A})| > |f(\mathbf{B})|$, then $\mathbf{B}$ is closer to the circle, so draw that pixel.
2. If $|f(\mathbf{A})| < |f(\mathbf{B})|$, then $\mathbf{A}$ is closer to the circle, so draw that pixel.
3. If $|f(\mathbf{A})| = |f(\mathbf{B})|$, the pixels are equidistant from the circle, so either one can be drawn.

The decision variable is $d = f(\mathbf{A}) + f(\mathbf{B})$. In Figure 2.34 (a), $f(\mathbf{A})$ and $f(\mathbf{B})$ are both negative, so $d < 0$. In part (c) of the figure, $f(\mathbf{A})$ and $f(\mathbf{B})$ are both positive,

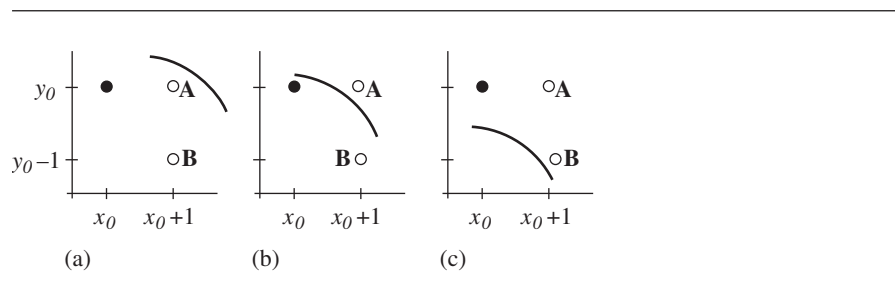


Figure 2.34 Deciding which circle pixel to draw next.

so $d > 0$. In part (b) of the figure, $f(\mathbf{A})$ is positive and $f(\mathbf{B})$ is negative. If $\mathbf{A}$ is closer to the circle than $\mathbf{B}$, then $|f(\mathbf{A})| < |f(\mathbf{B})|$ and so $d < 0$. If $\mathbf{B}$ is closer, then $|f(\mathbf{A})| > |f(\mathbf{B})|$ and $d > 0$. In all cases,

1. If $d > 0$, draw pixel $\mathbf{B}$.
2. If $d < 0$, draw pixel $\mathbf{A}$.
3. If $d = 0$, the pixels are equidistant from the circle, so draw pixel $\mathbf{A}$.

The current decision variable is constructed based on its previous value. Let

$$\begin{aligned} d_i &= (x_i + 1)^2 + y_i^2 - r^2 + (x_i + 1)^2 + (y_i - 1)^2 - r^2 \\ &= 2(x_i + 1)^2 + y_i^2 + (y_i - 1)^2 - 2r^2 \end{aligned}$$

Then

$$d_{i+1} - d_i = \begin{cases} 4x_i + 6, & y_{i+1} = y_i \\ 4x_i + 6 - 4y_i + 4, & y_{i+1} = y_i - 1 \end{cases}$$

The circle is centered at the origin. For a circle centered elsewhere, a simple translation of each pixel will suffice before drawing. Concise code is

```
void DrawCircle (int xcenter, int ycenter, int radius)
{
    for (int x = 0, y = radius, dec = 3-2*radius; x <= y; x++)
    {
        DrawPixel(xcenter+x,ycenter+y);
        DrawPixel(xcenter+x,ycenter-y);
        DrawPixel(xcenter-x,ycenter+y);
        DrawPixel(xcenter-x,ycenter-y);
        DrawPixel(xcenter+y,ycenter+x);
        DrawPixel(xcenter+y,ycenter-x);
        DrawPixel(xcenter-y,ycenter+x);
        DrawPixel(xcenter-y,ycenter-x);

        if ( dec >= 0 )
            dec += -4*(y--)+4;
        dec += 4*x+6;
    }
}
```

2.5.3 ELLIPSES

Rasterizing an ellipse is conceptually like rasterizing a circle, but the anisotropy of ellipses makes an implementation more challenging. The following material discusses how to conveniently specify the ellipse, how to draw an axis-aligned ellipse, and how to draw general ellipses.

Specifying the Ellipse

The algorithm described here draws ellipses of any orientation on a 2D raster. The simplest way for an application to specify the ellipse is by choosing an oriented bounding box with center (x_c, y_c) and axes (x_a, y_a) and (x_b, y_b) , where all components are integers. The axes must be perpendicular, so $x_a x_b + y_a y_b = 0$. It is assumed that (x_a, y_a) is in the first quadrant (not including the y -axis), so $x_a > 0$ and $y_a \geq 0$ are required. It is also required that the other axis is in the second quadrant, so $x_b \leq 0$ and $y_b > 0$. There must be integers n_a and n_b such that $n_b(x_b, y_b) = n_a(-y_a, x_a)$, but the algorithm does not require knowledge of these. The ellipse axes are the box axes and have the same orientation as the box.

All pixel computations are based on the ellipse with center $(0, 0)$. These pixels are translated by (x_c, y_c) to obtain the ones for the original ellipse. A quadratic equation for the ellipse centered at the origin is

$$\frac{(x_a x + y_a y)^2}{(x_a^2 + y_a^2)^2} + \frac{(x_b x + y_b y)^2}{(x_b^2 + y_b^2)^2} = 1$$

In this form it is easy to see that (x_a, y_a) and (x_b, y_b) are on the ellipse. Multiplying the matrices and multiplying through by denominators yields the quadratic equation

$$Ax^2 + 2Bxy + Cy^2 = D$$

where the integer coefficients are

$$A = x_a^2(x_b^2 + y_b^2)^2 + x_b^2(x_a^2 + y_a^2)^2$$

$$B = x_a y_a (x_b^2 + y_b^2)^2 + x_b y_b (x_a^2 + y_a^2)^2$$

$$C = y_a^2(x_b^2 + y_b^2)^2 + y_b^2(x_a^2 + y_a^2)^2$$

$$D = (x_a^2 + y_a^2)^2 (x_b^2 + y_b^2)^2$$

Since these integers can be quite large for standard-size rasters, an implementation should use 64-bit integers.

Axis-Aligned Ellipses

The algorithm for an axis-aligned ellipse draws the arc of the ellipse in the first quadrant and uses reflections about the coordinate axes to draw the other arcs. The ellipse centered at the origin is $b^2x^2 + a^2y^2 = a^2b^2$. Starting at $(0, b)$, the arc is drawn in clockwise order. The initial slope of the arc is 0. As long as the arc has a slope smaller than 1 in absolute magnitude, the x -value is incremented. The corresponding y -value is selected based on a decision variable, just as in Bresenham's circle-drawing algorithm. The remaining part of the arc in the first quadrant has a slope larger than 1 in absolute magnitude. That arc is drawn by starting at $(a, 0)$ and incrementing y at each step. The corresponding x -value is selected based on a decision variable.

While drawing the arc starting at $(0, b)$, let (x, y) be the current pixel that has been drawn. A decision must be made to select the next pixel $(x + 1, y + \delta)$ to be drawn, where δ is either 0 or -1 . The ellipse is defined implicitly as $Q(x, y) = 0$, where $Q(x, y) = b^2x^2 + a^2y^2 - a^2b^2$. Each choice for the next pixel lies on its own ellipse defined implicitly by $Q(x, y) = \lambda$ for some constant λ that is not necessarily zero. The idea is to choose δ so that the corresponding level curve has λ as close to zero as possible. This is the same idea that is used for Bresenham's circle algorithm. For the circle algorithm, the choice is based on selecting the pixel that is closest to the true circle. For ellipses, the choice is based on level set value and not on the distance between two ellipses (a much harder problem).

Given the current pixel (x, y) , for the next step the ellipse must do one of three things:

1. Pass below $(x + 1, y)$ and $(x + 1, y - 1)$, in which case $Q(x + 1, y) \geq 0$ and $Q(x + 1, y - 1) \geq 0$.
2. Pass between $(x + 1, y)$ and $(x + 1, y - 1)$, in which case $Q(x + 1, y) \geq 0$ and $Q(x + 1, y - 1) \leq 0$.
3. Pass above $(x + 1, y)$ and $(x + 1, y - 1)$, in which case $Q(x + 1, y) \leq 0$ and $Q(x + 1, y - 1) \leq 0$.

In the first case, the next pixel to draw is $(x + 1, y)$. In the second case, the pixel with Q value closest to zero is chosen. In the third case, the next pixel to draw is $Q(x + 1, y - 1)$. The decision in all three cases can be made by using the sign of $\sigma = Q(x + 1, y) + Q(x + 1, y - 1)$. If $\sigma < 0$, then the next pixel is $(x + 1, y - 1)$. If $\sigma > 0$, then the next pixel is $(x + 1, y)$. For $\sigma = 0$, either choice is allowed, so $(x + 1, y)$ will be the one selected.

The decision variable σ can be updated incrementally. The initial value is $\sigma_0 = Q(1, b) + Q(1, b - 1) = 2b^2 + a^2(1 - 2b)$. Given the current pixel (x, y) and decision variable σ_i , the next decision is

$$\sigma_{i+1} = \begin{cases} Q(x + 2, y) + Q(x + 2, y - 1), & \sigma_i \geq 0 \\ Q(x + 2, y - 1) + Q(x + 2, y - 2), & \sigma_i < 0 \end{cases}$$

The choice is based on whether or not the chosen pixel after (x, y) is $(x + 1, y)$ [when $\sigma_i > 0$] or $(x + 1, y - 1)$ [when $\sigma_i \leq 0$]. Some algebra leads to

$$\sigma_{i+1} = \sigma_i + \begin{cases} 2b^2(2x + 3), & \sigma_i \geq 0 \\ 2b^2(2x + 3) + 4a^2(1 - y), & \sigma_i < 0 \end{cases}$$

On this arc, x is always incremented at each step. The processing stops when the slope becomes 1 in absolute magnitude. The slope dy/dx of the ellipse can be computed implicitly from $Q(x, y) = 0$ as $Q_x + Q_y dy/dx = 0$, where Q_x and Q_y are the partial derivatives of Q with respect to x and y . Therefore, $dy/dx = -Q_x/Q_y = -(2b^2x)/(2a^2y) = -(b^2x)/(a^2y)$. The iteration on x continues as long as $-(b^2x)/(a^2y) \geq -1$. The termination condition of the iteration using only integer arithmetic is $b^2x \leq a^2y$.

The code for the iteration is

```
int a2 = a*a, b2 = b*b, fa2 = 4*a2;
int x, y, sigma;

for (x = 0, y = b, sigma = 2*b2+a2*(1-2*b); b2*x <= a2*y; x++)
{
    DrawPixel(xc+x,yc+y);
    DrawPixel(xc-x,yc+y);
    DrawPixel(xc+x,yc-y);
    DrawPixel(xc-x,yc-y);

    if ( sigma >= 0 )
    {
        sigma += fa2*(1-y);
        y--;
    }
    sigma += b2*(4*x+6);
}
```

The code for the other half of the arc in the first quadrant is symmetric in x and y and in a and b :

```
int a2 = a*a, b2 = b*b, fb2 = 4*b2;
int x, y, sigma;

for (x = a, y = 0, sigma = 2*a2+b2*(1-2*a); a2*y <= b2*x; y++)
{
    DrawPixel(xc+x,yc+y);
    DrawPixel(xc-x,yc+y);
    DrawPixel(xc+x,yc-y);
}
```

```

DrawPixel(xc-x,yc-y);

if ( sigma >= 0 )
{
    sigma += fb2*(1-x);
    x--;
}
sigma += a2*(4*y+6);
}

```

General Ellipses

An attempt could be made to mimic the case of axis-aligned ellipses by drawing the arc from (x_b, y_b) to (x_a, y_a) and reflecting each pixel (x, y) through the appropriate lines. For example, given pixel $\mathbf{u} = (x, y)$, the pixel reflected through $\mathbf{v} = (x_b, y_b)$ given by

$$(x', y') = \mathbf{u} - 2 \left(\frac{\mathbf{u} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}} \right) \mathbf{v} = (x, y) - 2 \left(\frac{x_b x + y_b y}{x_b^2 + y_b^2} \right) (x_b, y_b)$$

would also be drawn. The right-hand side requires a division. Moreover, even if the division is performed (whether as float or integer), the resulting pixels are not always contiguous and noticeable gaps occur. The general orientation of the ellipse requires a better method for selecting the pixels. Instead, the arc is generated from $(-x_a, -y_a)$ to (x_a, y_a) , and pixels $(x_c + x, y_c + y)$ and their reflections through the origin $(x_c - x, y_c - y)$ are plotted.

The algorithm is divided into two cases:

1. Slope at $(-x_a, -y_a)$ is larger than 1 in absolute magnitude. Five subarcs are drawn.
 - (a) Arc from $(-x_a, y_a)$ to a point (x_0, y_0) whose slope is infinite. For all points between, the ellipse has a slope larger than 1 in absolute magnitude, so y is always incremented at each step.
 - (b) Arc from (x_0, y_0) to a point (x_1, y_1) whose slope is 1. For all points between, the ellipse has a slope larger than 1 in absolute magnitude, so y is always incremented at each step.
 - (c) Arc from (x_1, y_1) to a point (x_2, y_2) whose slope is 0. For all points between, the ellipse has a slope less than 1 in absolute magnitude, so x is always incremented at each step.
 - (d) Arc from (x_2, y_2) to a point (x_3, y_3) whose slope is -1 . For all points between, the ellipse has a slope less than 1 in absolute magnitude, so x is always incremented at each step.

- (e) Arc from (x_3, y_3) to (x_a, y_a) . For all points between, the ellipse has a slope larger than 1 in absolute magnitude, so y is always decremented at each step.
- 2. Slope at $(-x_a, -y_a)$ is smaller than 1 in absolute magnitude. Five subarcs are drawn.
 - (a) Arc from $(-x_a, -y_a)$ to a point (x_0, y_0) whose slope is -1 . For all points between, the ellipse has a slope less than 1 in absolute magnitude, so x is always decremented.
 - (b) Arc from (x_0, y_0) to a point (x_1, y_1) whose slope is infinite. For all points between, the ellipse has a slope larger than 1, so y is always incremented.
 - (c) Arc from (x_1, y_1) to a point (x_2, y_2) whose slope is 1. For all points between, the ellipse has a slope larger than 1 in absolute magnitude, so y is always incremented at each step.
 - (d) Arc from (x_2, y_2) to a point (x_3, y_3) whose slope is 0. For all points between, the ellipse has a slope less than 1 in absolute magnitude, so x is always incremented at each step.
 - (e) Arc from (x_3, y_3) to (x_a, y_a) . For all points between, the ellipse has a slope less than 1 in absolute magnitude, so x is always incremented at each step.

Each subarc is computed using a decision variable as in the case of an axis-aligned ellipse. The decision to switch between the three subarcs is based on the slope of the ellipse. The ellipse is implicitly defined by $Q(x, y) = 0$, where $Q(x, y) = Ax^2 + 2Bxy + Cy^2 - D = 0$. The derivative $dy/dx = -(Ax + By)/(Bx + Cy)$ is obtained by implicit differentiation. The numerator and denominator of the derivative can be maintained incrementally. Initially, the current pixel $(x, y) = (-x_a, -y_a)$ and the numerator and denominator of the slope are $dy = Ax_a + By_a$ and $dx = -(Bx_a + Cy_a)$.

The decision variable σ is handled slightly differently than in the case of an axis-aligned ellipse. In the latter case, the decision was made to use the pixel whose own level curve is closest to the zero level curve. In the current case, a general ellipse handled in the same way can lead to gaps at the endpoints of the arc and the reflected arc. To avoid the gaps, the decision is made to always select the ellipse with the *smallest positive* level curve value rather than the smallest magnitude level curve value. The selected pixels are always outside the true ellipse. The decision variable is not incrementally maintained because it is not expensive to compute, although it is possible to maintain it so.

Each of the algorithms for the 10 subarcs are similar in structure. Case 1(a) is described here. The initial values are $x = -x_a$, $y = -y_a$, $dx = Bx_a + Cy_a$, and $dy = -(Ax_a + By_a)$. As y is incremented, eventually the leftmost point in the x -direction is encountered where the slope of the ellipse is infinite. At each step the two pixels to test are $(x, y + 1)$ and $(x - 1, y + 1)$. It is enough to test $\sigma = Ax^2 + 2Bx(y + 1) + C(y + 1)^2 - D < 0$ to see if $(x, y + 1)$ is inside the true ellipse. If it is, then $(x - 1, y + 1)$ is the next pixel to draw. If $\sigma \geq 0$, then $(x, y + 1)$ is outside the

true ellipse and closer to it than $(x - 1, y + 1)$, so the next pixel is $(x, y + 1)$. The code is

```
while ( dx <= 0 ) // Loop until point with infinite slope occurs.
{
    DrawPixel(xc+x,yc+y);
    DrawPixel(xc-x,yc-y);
    y++;
    sigma = a*x*x+2*b*x*y+c*y*y-d;
    if ( sigma < 0 )
    {
        dx -= b;
        dy += a;
        x--;
    }
    dx += c;
    dy -= b;
}
```

The other nine cases are structured similarly.

2.5.4 TRIANGLES

Drawing a triangle as a white object on a black background is a simple process that determines the pixels with minimum and maximum x -values on each scan line intersected by the triangle, then draws the pixels between. This is accomplished by keeping two buffers for the minimum and maximum, with each buffer having a number of elements equal to the height of the screen, and using the Bresenham line-drawing algorithm to draw the three edges of the triangle. The line drawer updates the buffers when necessary. It is useful to sort the vertices on y so that the line drawer can update only one of the buffers at a time. This also helps to trap degenerate triangles that are passed to the rasterizer; the degeneracy is caused by triangles seen nearly edge on by the eye point, with numerical round-off errors leading to the projection being a line segment. Pseudocode is given for a triangle with integer-valued vertices (x_i, y_i) for $0 \leq i \leq 2$ that are listed in counterclockwise order. There are 13 cases, six of the form $y_{i_0} < y_{i_1} < y_{i_2}$, three of the form $y_{i_0} = y_{i_1} < y_{i_2}$, three of the form $y_{i_0} < y_{i_1} = y_{i_2}$, and one of the form $y_{i_0} = y_{i_1} = y_{i_2}$. Only a couple of the cases are listed in the pseudocode. It is assumed that there are two update routines, one that updates the minimum buffer (UpdateMin) and one that updates the maximum buffer (UpdateMax). The return value of false indicates a degenerate triangle, true otherwise.

```
// global quantities
xmin[0..H-1] = minimum x-values for scan lines  $0 \leq y \leq H-1$ ;
xmax[0..H-1] = maximum x-values for scan lines  $0 \leq y \leq H-1$ ;
```



```

ymin = last minimum y-value for scan lines;
ymax = last maximum y-value for scan lines;
pixel[0..H-1][0..W-1] = frame buffer;

bool ComputeEdgeBuffers ()
{
    /*** case: y0 < y1 < y2
    dx0 = x1-x0; dy0 = y1-y0; dx1 = x2-x0; dy1 = y2-y0;
    det = dx0*dy1-dx1*dy0;
    // Assert: det <= 0 since vertices are counterclockwise and
    // window space has left-handed coordinates.
    if (det < 0)
    {
        UpdateMin(x0,y0,x1,y1);
        UpdateMin(x1,y1,x2,y2);
        UpdateMax(x0,y0,x2,y2);
        return true;
    }
    else
    {
        // degenerate triangle
        return false;
    }

    /*** case: y0 < y1 = y2
    // Assert: x1 <= x2 since vertices are counterclockwise and
    // window space has left-handed coordinates.
    if (x1 < x2)
    {
        UpdateMax(x0,y0,x2,y2);
        UpdateMin(x0,y0,x1,y1);
        return true;
    }
    else
    {
        // degenerate triangle
        return false;
    }
}

```

Lines are always drawn starting from the vertex with the smaller y-value. This avoids the cracking between triangles that was mentioned in Section 2.5.1. The triangle rasterizer is

```

void DrawWhiteTriangle ()
{
    clear xmin[ymin..ymax];
    clear xmax[ymin..ymax];

    if (ComputeEdgeBuffers())
    {
        for (y = ymin; y <= ymax; y++)
        {
            for (x = xmin[y]; x <= xmax[y]; x++)
                pixel[y][x] = WHITE;
        }
    }
}

```

The actual code in the Wild Magic software renderer is slightly more complicated. You will find this in the file `Wm4SoftDrawElements.cpp`, function `SoftRenderer::RasterizeTriangle`. The input triangle has vertices in clip-space coordinates, which were computed in the vertex shader program. These coordinates are converted to NDC values, and then the perspective divide is applied to produce window coordinates. Before the `ComputeEdgeBuffers` function is called, a check is made to see if the triangle is front facing. Although we already tested for this earlier in the process—before clipping occurs—it is possible that numerical round-off errors produce clipped triangles that are back facing. If the input triangle is back facing, the rasterizer simply returns without drawing.

If the input triangle is front facing, `ComputeEdgeBuffers` is called. The assignment of the pixel color in the inner loop of the pseudocode previously mentioned is an oversimplification. For each pixel (x, y) , the rasterizer must interpolate vertex attributes, this being done in clip-space coordinates, and then convert to NDC values and project to window coordinates. At that time the pixel is “drawn” by calling the entry function to the pixel shading system. The actual inner loop is

```

PerspectiveInterpolate(afXMinAttr,afXMaxAttr,iX0,iX1,m_aafScanAttr);
for (int iX = iX0 + 1; iX < iX1; iX++)
{
    ClipToWindow(m_aafScanAttr[iX],fXWindow,fYWindow,fDepth,fInverseW);
    ApplyPixelShader(iIndex++,fDepth,fInverseW,&m_aafScanAttr[iX][4]);
}

```

The variable `iX0` corresponds to the pseudocode value `xmin[y]`, and the variable `iX1` corresponds to the pseudocode value `xmax[y]`. The arrays `afXMinAttr` and `afXMaxAttr` store vertex attributes such as colors and texture coordinates. The first line of the actual code interpolates all the vertex attributes at the endpoints of the scan line of the triangle, using perspective interpolation (not linear interpolation), and the results

are stored in the data structure `m_aafScanAttr`. The inner loop follows next, where the function `ClipToWindow` converts clip-space coordinates to NDC values, and then to window coordinates. The function `ApplyPixelShader` is the entry point to actually drawing the pixel, but involves additional logic related to depth buffering, stencil buffering, alpha blending, and color masking. The details on the vertex and pixel shader systems are found in Chapter 3.

2.6 VERTEX ATTRIBUTES

Triangles are drawn by the renderer as colored entities, the color of each pixel determined by *vertex attributes* assigned to the vertices of the triangle. The pixels at nonvertex locations are computed via interpolation by the rasterizer, the final values in total called *surface attributes*. In screen space the projected vertices have locations (x, y) that are used to control the interpolation process. Each vertex is endowed with a list of attributes depending on how the application wants the triangle to be drawn.

2.6.1 COLORS

Each vertex can be assigned a *vertex color* $C = (r, g, b, a)$, where r is the red channel, g is the green channel, b is the blue channel, and a is the alpha channel (for transparency effects). Channels from other color models could be used instead, especially now that vertex shader programming is in your control. A rasterized triangle whose vertices are assigned only colors is not that visually appealing, since interpolation of three color values over a triangle does not produce a wide variation in color. However, using only vertex colors may be necessary either on systems with a limited amount of memory, which prevents having a large number of textures at hand, or on systems with slow processors that take many cycles to combine multiple colors. Vertex colors are typically used in conjunction with textures to add more realism to the rendering. Moreover, the vertex colors can be used in conjunction with lights in the scene to generate dynamic effects, such as a flaming fireball traveling down a corridor and lighting portions of the walls near its path. This is termed *dynamic lighting* and is described in the next section.

2.6.2 LIGHTING AND MATERIALS

Dynamic lighting effects can be achieved by using light sources to illuminate portions of the scene and by assigning material properties to various objects in the scene.

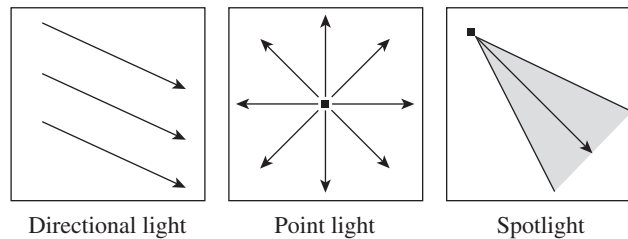


Figure 2.35 Various light sources.

Lights

The standard light sources in a real-time engine are

- Directional lights. The light source is assumed to be infinitely far away so that the directions of the light rays are all parallel. The sun is the classic example of a directional light.
- Point lights. The light source has a location in space and emits light in all directions.
- Spotlights. The light source has a location in space but emits light only within a cone.

Figure 2.35 illustrates the three possible sources. Real light sources emit light from an area or volume source. Point light sources are a reasonable approximation in a real-time setting but do not always produce visually correct information. For example, shadows generated by a point source have hard edges, but shadows generated by a real light source have soft edges.

Engines also have the concept of *ambient light*. This is not really a light source, but amounts to contributions from scattered light in the surrounding environment to the illumination of an object. Throughout the discussion here, I will refer to the four light types: ambient, directional, point, and spot.

Light sources have various attributes in addition to position and direction. Each light can be monochrome or can have an RGB color associated with it. Instead of a single color for the light, multiple colors can be used to represent the contribution to ambient, diffuse, and specular lighting. The light can also maintain an intensity parameter that applies to the various colors, and a Boolean parameter can be used to indicate whether the light is on or off, a quick way to enable or disable lights in the rendering system. Other attributes assigned to lights depend on type. Point lights and

spotlights can have their light attenuated with distance from the light source, with the attenuation parameter usually specified as an inverse quadratic:

$$\alpha = \begin{cases} I, & \text{Ambient and directional lights} \\ \frac{I}{c_0 + c_1|\mathbf{V} - \mathbf{P}| + c_2|\mathbf{V} - \mathbf{P}|^2}, & \text{Point lights and spotlights} \end{cases} \quad (2.76)$$

where $\mathbf{P}$ is the light position and $\mathbf{V}$ is a point to be illuminated. The coefficients are I for intensity, c_0 for the constant term, c_1 for the linear term, and c_2 for the quadratic term. The physically correct model is $I = 1$ and $c_0 = c_1 = 0$ and produces the inverse square relationship that we expect. However, the c_0 and c_1 parameters give an application more control over how the attenuation is to occur. Moreover, choosing $c_0 > 0$ guards against floating-point overflow when $|\mathbf{V} - \mathbf{P}|$ is nearly zero. Usually, the models for attenuation distance always have $I = 1$, but I include the coefficient I in the Wild Magic implementation of lights to allow for an intuitive linear adjustment of intensity. Without it, you can adjust the intensity by varying c_0 , but linear changes in c_0 do not equate to linear changes in intensity.

Materials

Associating a material with an object is an attempt to give the object surface characteristics based on the material parameters and the light sources. The material parameters include emissive M_{emis} , ambient M_{ambi} , diffuse M_{diff} , and specular M_{spec} color components and include scalar parameters for shininess M_{shine} and alpha blending M_{alpha} . The emissive color represents the fact that a material itself can emit light rather than simply reflect it. The ambient, diffuse, and specular colors are intended to be factors that interact with the light sources. Shininess is used to control how sharp or diffuse a specular highlight is. The alpha value is used to support transparent materials as an alternative to applying texture images with an alpha channel.

Lighting and Shading

The term *lighting* refers to the process of computing colors based on light sources and materials. The term *shading* refers to the process of computing pixel colors after any lighting has been calculated. The three standard shading models are *flat*, *Gouraud*, and *Phong*. Flat shading uses the same color for all pixels in a rendered triangle. Thus, a color is assigned to the entire triangle rather than separate colors assigned to the three vertices. Gouraud shading calculates the vertex colors of the triangle based on light sources and materials, then interpolates those colors to fill out the remaining pixels in the triangle. Phong shading takes the three vertex normals and interpolates them to compute a normal vector per pixel. Each pixel is then lit according to the light sources and materials that affect the triangle. The fixed-function graphics pipeline supports flat shading and Gouraud shading, but Phong shading is more expensive to

compute and not directly supported on consumer machines. With a programmable graphics pipeline, it is possible to use Phong shading with normal vector interpolation accomplished through what are called *normalization maps*. The normal vectors are linearly interpolated during rasterization but made unit length in a pixel shader by a lookup into a texture specifically designed for this purpose.

In a software renderer, it is possible to interpolate normal vectors in a nonlinear manner. Specifically, if the three vertex normals are plotted on a unit sphere, the normal at any triangle interior point corresponds to a point on the unit sphere contained in the spherical triangle formed by the original three normals. A discretization of the spherical triangle is quite possible and not expensive ([And94, AJ97]), but this would have to be part of the rasterization in the graphics system, something that is not exposed to the developer through shader programming.

The colors at the triangle vertices are computed via a *lighting model*. The models used in real-time graphics involve decomposition into ambient, diffuse, and specular components. The model described here assumes that each light has an ambient color $\mathbf{L}_{\text{ambi}}$, a diffuse color $\mathbf{L}_{\text{diff}}$, a specular color $\mathbf{L}_{\text{spec}}$, and an intensity L_{intn} that is applied equally to all three colors. Point lights and spotlights also have an attenuation value L_{attn} .

The lighting contributions fall into three categories: ambient, diffuse, and specular. In the discussions here, I omit mention of the light intensity, but in the final discussion about adding up all the contributions (the lighting equation), I incorporate this quantity.

Ambient Lighting

A light ray in the real world follows a path that has it reflecting off many surfaces and decreasing in intensity along the way. The global effect from all the rays is termed *ambient lighting*. The light model incorporates this effect by combining the light's ambient color with the material's ambient color,

$$\mathbf{C}_{\text{ambi}} = \mathbf{M}_{\text{ambi}} \circ \mathbf{L}_{\text{ambi}} \quad (2.77)$$

where the operator $\circ$ represents componentwise multiplication. The color channels are chosen to be values in the interval $[0, 1]$. If two color channels are in this interval, the product is in the interval. If you adjust the intensity parameter to be large enough to cause a color channel to be larger than 1, you typically will clamp the value to 1, but effects using *high dynamic range lighting* do not require such clamping. A good book on high dynamic range imaging is [RWPD05].

Diffuse Lighting

Diffuse lighting is based on Lambert's law, which says that for a matte surface, the intensity of the reflected light is determined by the cosine of the angle between the outer-pointing surface normal $\mathbf{N}$ and the light direction vector $\mathbf{D}$. Moreover, the

intensity drops to zero when the angle between $\mathbf{N}$ and $\mathbf{D}$ is $\pi/2$ radians or smaller. The light model incorporates diffuse lighting by

$$\mathbf{C}_{\text{diff}} = \mu \mathbf{M}_{\text{diff}} \circ \mathbf{L}_{\text{diff}} \quad (2.78)$$

where

$$\mu = \max\{-\mathbf{N} \cdot \mathbf{D}, 0\} \quad (2.79)$$

is the *diffuse coefficient*. The light direction depends on the light type. For directional lights, the direction $\mathbf{D}$ is already known. For point lights and spotlights, the direction is $\mathbf{D} = (\mathbf{V} - \mathbf{P})/|\mathbf{V} - \mathbf{P}|$, where the light source location is $\mathbf{P}$ and the point $\mathbf{V}$ is to be illuminated.

Specular Lighting

Diffuse lighting represents reflection of light on matte surfaces. Specular lighting represents reflection of light on shiny surfaces. In particular, specular highlights can show up on highly reflective surfaces. These are places where the surface normal and light direction are parallel. The tightness of the region of brightness is something that can be controlled by the material's shininess parameter.

The Phong lighting equation [Pho00] is a model for computing the specular lighting. Figure 2.36 (a) illustrates the various point and vector quantities in the model. Let $\mathbf{E}$ be the eye point. Let $\mathbf{U} = (\mathbf{V} - \mathbf{E})/|\mathbf{V} - \mathbf{E}|$ be the view direction for a point $\mathbf{V}$ to be illuminated. Let $\mathbf{D}$ be the light direction, specified for directional lights but computed to be $\mathbf{D} = (\mathbf{V} - \mathbf{P})/|\mathbf{V} - \mathbf{P}|$ for point lights and spotlights. The reflection vector of the light direction through the surface normal $\mathbf{N}$ is

$$\mathbf{R} = \mathbf{D} - 2(\mathbf{N} \cdot \mathbf{D})\mathbf{N}$$

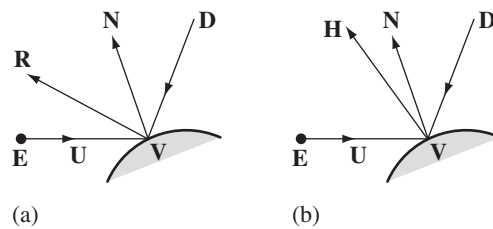


Figure 2.36 (a) The Phong lighting model. The angle between the reflection vector $\mathbf{R}$ and the surface normal $\mathbf{N}$ controls the attenuation. (b) The Blinn lighting model. The angle between the half vector $\mathbf{H}$ and the surface normal $\mathbf{N}$ controls the attenuation.

If the observer looks at $\mathbf{V}$ in the direction $\mathbf{U} = -\mathbf{R}$, he should see a very bright region at the point. As the view direction varies away from $-\mathbf{R}$, the region at the point should be less bright. The specular coefficient for the attenuation is chosen to be

$$\gamma = (\max\{-\mathbf{R} \cdot \mathbf{U}, 0\})^{M_{\text{shine}}} \quad (2.80)$$

The clamping to zero means that an observer behind the surface will see no reflected light. The exponent M_{shine} controls how small (or large) the bright region is around $\mathbf{V}$. The specular lighting is

$$\mathbf{C}_{\text{spec}} = \gamma \mathbf{M}_{\text{spec}} \circ \mathbf{L}_{\text{spec}} \quad (2.81)$$

The Blinn lighting equation [Bli77] is a variation of the Phong lighting equation. Figure 2.36 (b) illustrates this model. The *half vector* $\mathbf{H}$ is chosen to be

$$\mathbf{H} = -\frac{\mathbf{D} + \mathbf{U}}{|\mathbf{D} + \mathbf{U}|}$$

If the observer looks at $\mathbf{V}$ in the direction $\mathbf{U} = -\mathbf{H}$, he should see a very bright region at the point. As the view direction varies away from $-\mathbf{H}$, the region at the point should be less bright. The specular coefficient for the attenuation is chosen to be

$$\gamma = (\max\{-\mathbf{R} \cdot \mathbf{H}, 0\})^{M_{\text{shine}}} \quad (2.82)$$

The variation was designed to produce similar visual results, but at a smaller computational cost when a directional light is used and a *nonlocal observer* is assumed. The latter term means that the observer is sufficiently far away from the object that the view direction $\mathbf{U}$ may be chosen to be a constant direction. Thus, $\mathbf{H}$ is computed once and used for diffuse lighting of all the vertices on a surface. When the view direction varies, $\mathbf{H}$ must be computed per vertex.

Vertex shader programming allows you to implement whatever model you prefer. The lighting vertex shaders in Wild Magic use the Blinn model and a *local viewer*, so $\mathbf{H}$ is computed per vertex.

Spot Attenuation

Spotlights have a position $\mathbf{P}$, a unit-length cone axis $\mathbf{A}$, and an angle $\theta \in (0, \pi/2)$ that is measured from the cone axis to the cone wall. A vertex $\mathbf{V}$ is illuminated by a spotlight only when it is inside the cone of the spotlight. The light direction to the vertex is $\mathbf{D} = (\mathbf{V} - \mathbf{P})/|\mathbf{V} - \mathbf{P}|$. Attenuation based solely on containment in the cone is controlled by

$$\sigma = \begin{cases} 1, & \mathbf{D} \cdot \mathbf{A} \geq \cos \theta \\ 0, & \text{otherwise} \end{cases} \quad (2.83)$$

With this form of attenuation, a coarsely tessellated mesh will have jagged edges separating the lit portion of the surface from the nonlit portion, an unattractive visual artifact. To attempt to eliminate this, you may also allow the intensity to decrease as the angle from the cone axis increases,

$$\sigma = \begin{cases} (\mathbf{D} \cdot \mathbf{A})^{L_{\text{spot}}}, & \mathbf{D} \cdot \mathbf{A} \geq \cos \theta \\ 0, & \text{otherwise} \end{cases} \quad (2.84)$$

where $L_{\text{spot}} > 0$ is referred to as the *spot exponent*. This helps remove some of the jaggedness, but the light intensity at the cone edge is still discontinuous. An even more expensive form of attenuation is to have a factor that is 1 when $\mathbf{V}$ is on the cone axis, but then decreases smoothly to 0 when $\mathbf{V}$ is on the cone wall,

$$\sigma = \begin{cases} \left(\frac{\mathbf{D} \cdot \mathbf{A} - \cos \theta}{1 - \cos \theta} \right)^{L_{\text{spot}}}, & \mathbf{D} \cdot \mathbf{A} \geq \cos \theta \\ 0, & \text{otherwise} \end{cases} \quad (2.85)$$

The Wild Magic vertex shaders for lighting use the spot attenuation of Equation (2.84). If you choose the spot exponent sufficiently large, the jaggedness artifacts are no longer visible.

The Lighting Equation

The final equation for lighting a vertex with a single material and using multiple light sources, which follows, includes the attenuation factors for distance as well as for spot angles. The superscripts are indices for the array of active lights.

$$\begin{aligned} \mathbf{C}_{\text{final}} = \mathbf{M}_{\text{emis}} & \quad (2.86) \\ + \sum_{i=1}^n \alpha_i \sigma_i & \left(\mathbf{M}_{\text{ambi}} \circ \mathbf{L}_{\text{ambi}}^i + \mu_i \mathbf{M}_{\text{diff}} \circ \mathbf{L}_{\text{diff}}^i + \gamma_i \mathbf{M}_{\text{spec}} \circ \mathbf{L}_{\text{spec}}^i \right) \end{aligned}$$

where α_i is the distance attenuation for the i th light, computed using Equation (2.76); σ_i is the spot attenuation, which is 1 for point or directional lights but computed for spotlights using one of Equations (2.83), (2.84), or (2.85); μ_i is the diffuse coefficient, computed using Equation (2.79); and γ_i is the specular coefficient, computed using either Equation (2.80) or (2.82). The specular coefficient and spot attenuation are set to zero whenever the diffuse coefficient is zero because the observer cannot see the lit vertex.

The actual shader programs use a factored version of Equation (2.86) to allow for code fragments to handle any number of lights. This reduces the number of computations that the shader program must perform. The factored equation is

$$\begin{aligned}
\mathbf{C}_{\text{final}} = & \mathbf{M}_{\text{emis}} + \mathbf{M}_{\text{ambi}} \circ \sum_{i=1}^n \alpha_i \sigma_i \mathbf{L}_{\text{ambi}}^i \\
& + \mathbf{M}_{\text{diff}} \circ \sum_{i=1}^n \alpha_i \sigma_i \mu_i \mathbf{L}_{\text{diff}}^i + \mathbf{M}_{\text{spec}} \circ \sum_{i=1}^n \alpha_i \sigma_i \gamma_i \mathbf{L}_{\text{spec}}^i
\end{aligned} \tag{2.87}$$

Note that if no lights are present and the material emits light, the final vertex color is not black. It is also possible to include a global ambient light term $\mathbf{M}_{\text{ambi}} \circ \mathbf{G}_{\text{ambi}}$, where the global ambient color is specified by the application, but I do not include this in Wild Magic. You simply attach an ambient light object to the root of the scene to obtain a global ambient light.

2.6.3 TEXTURES

Textured images, or simply *textures*, provide the most realism in a model and can be used effectively to hide the model's polygonal aspects. Although in most cases the images are 2D, there are many special effects that use 1D and 3D images. A *texture coordinate* is a 1-tuple (s) for a 1D image, a 2-tuple (s, t) for a 2D image, and a 3-tuple (s, t, r) for a 3D image. Advanced special effects can use *projected textures*, in which case the texture coordinate is a homogeneous 4-tuple (s, t, r, q). The actual values used for looking up the color in the image are obtained by the perspective divide: s/q for a 1D image, $(s/q, t/q)$ for a 2D image, and $(s/q, t/q, r/q)$ for a 3D image. The texture coordinate can have an even more complicated representation; for example, shadow maps use *depth textures*. The r -values represent distances from a light source, and the $(s/q, t/q)$ -values are used for a lookup into a texture representing depth values. Some of the sample shaders discussed in Chapter 20 have the details on how the texture coordinates are manipulated. For general notation, if $\mathbf{u}$ is a texture coordinate (in whatever dimension), the texture image has a corresponding color value $\mathbf{C}(\mathbf{u})$.

A mesh of triangles may be assigned a textured image and each vertex may be assigned a texture coordinate. The texture coordinates at the vertices are perspectively interpolated by the rasterizer to obtain texture coordinates at other pixels in the triangle. Each interpolated coordinate is also used to do a color lookup in the image. Since the components of the texture image are, in practice, integer valued, and since the interpolation produces floating-point values that are not necessarily integers, the lookup of the corresponding color in the texture image requires an interpretation of the texture coordinate. This is the subject of *coordinate modes*, *filter modes*, and *mipmap modes*.

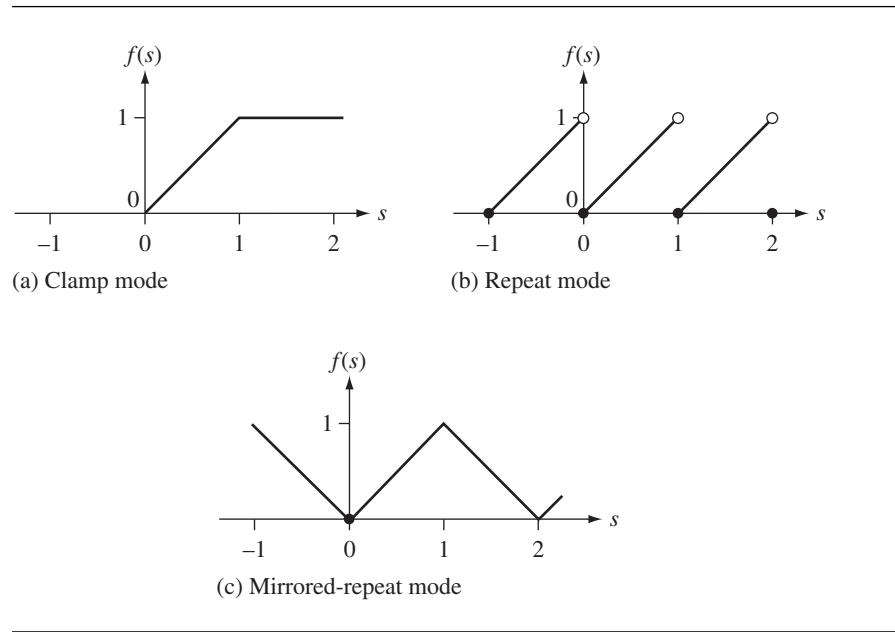


Figure 2.37 (a) The graph of the clamping function. (b) The graph of the repeat function. (c) The graph of the mirrored-repeat function.

Coordinate Modes

The standard representation of a 2D texture image uses texture coordinates that have components in the interval $[0, 1]$. The images have their own bounds per dimension, and the lookup of the color in the image requires some type of transformation from $[0, 1]$ to the image bound. That said, it is not necessary that a texture coordinate have components in $[0, 1]$. This allows for efficient use of textures and for interesting effects. The standard texture coordinate modes are *clamping*, *repeating* (or *wrapping*), and *mirrored repeating*. Each of these modes is applied per component of a texture coordinate.

A coordinate s is *clamped* by setting

$$s' = f(s) = \min\{\max\{0, s\}, 1\} \quad (2.88)$$

That is, if $s < 0$, then $s' = 0$, or if $s > 1$, then $s' = 1$; otherwise, $s' = s$. Figure 2.37 (a) shows a graph of the clamp function.

One special effect obtained by clamping is to place a small detail in the interior of a triangle. For example, a triangle that represents part of a glass window can have a texture applied to make it appear as if the window has a bullet hole in it. The texture

image for the bullet hole can be quite small (to minimize memory usage), and the texture coordinates for the vertices can be set to quantities well outside the range of $[0, 1]^2$ to control the size and placement of the bullet hole.

A coordinate s is *repeated* by setting

$$s' = f(s) = s - \lfloor s \rfloor \quad (2.89)$$

where $\lfloor s \rfloor$ is the largest integer smaller than or equal to s . Figure 2.37 (b) shows a graph of the repeat function. The typical special effect obtained by wrapping is to allow a texture to repeat, thereby producing a doubly periodic effect. The texture in this case is said to be *toroidal*, and great care must be taken so that the left/right edges and top/bottom edges of the texture match (otherwise the texture boundaries are noticeable) in the replication. For example, a brick wall can be built from a small number of triangles with a small texture representing a few bricks.

A coordinate s is *mirror repeated* by setting

$$s' = f(s) = \begin{cases} s - \lfloor s \rfloor, & \lfloor s \rfloor \text{ is even} \\ 1 - (s - \lfloor s \rfloor), & \lfloor s \rfloor \text{ is odd} \end{cases} \quad (2.90)$$

Figure 2.37 (c) shows a graph of the mirrored-repeat function. Whether you use repeat or mirror-repeat modes, the texture image values should match whenever $s = 0$ and $s = 1$; otherwise, you have a visual anomaly called a *seam*. For a 2D image, if the image values satisfy $C(0, t) = C(1, t)$ for all $t \in [0, 1]$, but $C(s, 0) \neq C(s, 1)$ for some $s \in [0, 1]$, the texture is said to be *cylindrical*. The idea is that if you wrap the image and connect two opposing sides, you have a cylinder and the image varies smoothly as you walk around the cylinder. If instead you have $C(0, t) = C(1, t)$ for all $t \in [0, 1]$ and $C(s, 0) = C(s, 1)$ for all $s \in [0, 1]$, the texture is said to be *toroidal*. The idea is that if you wrap the image and connect two opposing sides, then connect the other two opposing sides, you have a torus and the image varies smoothly as you walk around the torus.

Figure 2.38 shows a texture that is clamped in the horizontal direction, Figure 2.39 shows a texture that is repeated in the horizontal direction, and Figure 2.40 shows a texture that is mirror repeated in the horizontal direction. The texture image is defined on a rectangular lattice of points, so it is not a continuous quantity. A texture coordinate computed via interpolation is usually not a lattice point. The method of computing a lattice point for the coordinate is called *texture filtering*, but is also referred to as *texture sampling*. A fundamental issue in sampling is deciding how to interpret what a texture image really is. The image consists of a discrete collection of *point* samples, but the interpolation needs a continuous representation. My illustrations here are for 1D images, but the concepts trivially extend to higher-dimensional images.

Let the image consist of N color samples, C_i for $0 \leq i \leq N - 1$. The standard interpretation in computer graphics is to think of the samples as the *centers* of a line



Figure 2.38 A 2D texture clamped in the horizontal direction.

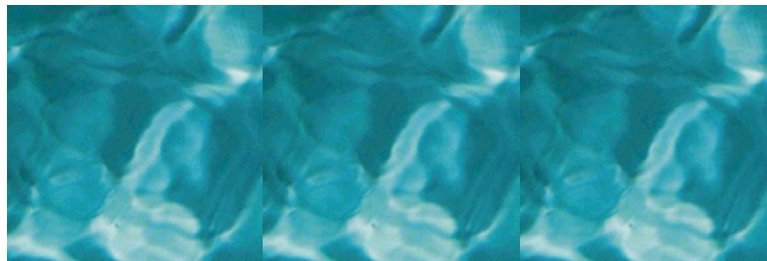


Figure 2.39 A 2D texture repeated in the horizontal direction.

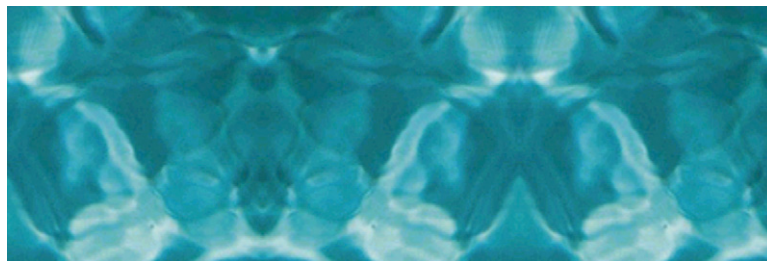


Figure 2.40 A 2D texture mirror repeated in the horizontal direction.

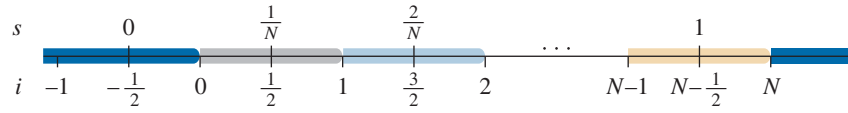


Figure 2.41 A 1D texture image. Each texel represents an interval on the real line, with each point in that interval assigned the texture image value. The centers of the texels are the centers of the line segments. The texture coordinates are listed above the centers.

segment (of a square in 2D and of a cube in 3D).¹ Figure 2.41 illustrates this for a 1D image. The texel centers are at image indices $i = -1/2, 1/2, 3/2, \dots, N - 1/2$. The sample value C_0 is shown as gray. This color is assigned to all (continuous) index values in the interval $[0, 1)$. This interval includes the index 0 but excludes the index 1. The sample value C_1 is shown as light blue. This color is assigned to all index values in the interval $[1, 2)$, which includes 1 but excludes 2. The last sample value C_{N-1} is shown as taupe and is assigned to all index values in the interval $[N - 1, N)$, which includes $N - 1$ but excludes N . Indices might fall outside the domain of the image, in which case a *border color* can be specified as the texture color. In the figure, the border color is shown as dark blue. If an index i has the property that $i < 0$ or $i \geq N$, the border color is used as the texture image lookup value. The relationship between the texture coordinate s and the continuous image index i is $i = Ns - 1/2$.

With our choice of texel centers, the continuous image index i is computed from the texture coordinate s by the following formulas. For clamp mode, s' is computed using Equation (2.88). The image index i corresponding to s is $i = Ns - 1/2$. The image index i' corresponding to s' is computed using

$$i' = \begin{cases} -1/2, & i < -1/2 \\ i, & -1/2 \leq i \leq N - 1/2 \\ N - 1/2, & i > N - 1/2 \end{cases} \quad (2.91)$$

The range of i' values is $[-1/2, N - 1/2]$.

Graphics APIs have provided a couple of variations on clamp mode. The first variation is the *clamp-to-edge* mode. Rather than clamp to the range $[-1/2, N - 1/2]$,

1. This is not always the interpretation in other fields. For example, in medical image analysis, the construction of a level curve in a 2D image involves treating a pixel as a square whose corners are four image samples. The construction of a level surface in a 3D image involves treating a voxel as a cube whose corners are eight image samples. Some computer graphics practitioners would disagree with such an interpretation.

the values are clamped to be valid image indices, namely, $[0, N - 1]$. The formula for clamping to an edge is

$$i' = \begin{cases} 0, & i < 0 \\ i, & 0 \leq i \leq N - 1 \\ N - 1, & i > N - 1 \end{cases} \quad (2.92)$$

This mode avoids having to use a border color because the indices are never out of range. The second variation is the *clamp-to-border* mode. The range of clamped values is $[-1, N]$. The formula for clamping to the border is

$$i' = \begin{cases} -1, & i < -1 \\ i, & -1 \leq i \leq N \\ N - 1, & i > N \end{cases} \quad (2.93)$$

For repeat mode or mirrored-repeat mode, s' is computed using either Equation (2.89) or (2.90). The image index i' corresponding to s' is computed using

$$i' = Ns' - 1/2 \quad (2.94)$$

The range of the index is $[-1/2, N - 1/2]$.

All texture coordinate modes produce the same value of i' for a specified s when the image index lies in the interior of the image. The differences in sampling occur only at the boundary. How they differ depends on the *filtering mode*, which is discussed next.

Filtering Modes

There are two standard ways to filter an image. The first method takes the continuous image index i' , computed using one of Equations (2.91) through (2.94), and selects the *nearest* integer index $j = \lfloor i' + 1/2 \rfloor$. The image value $\mathbf{C}$ used for the pixel color is

$$\mathbf{C} = \mathbf{C}_j \quad (2.95)$$

This gives a jagged appearance, especially when the texture image is high frequency in its data. Figure 2.42 illustrates this for a 2D texture with a checkerboard image.

The second method uses *linear interpolation* (1D images), *bilinear interpolation* (2D images), or *trilinear interpolation* (3D images) as a way of smoothing the results and avoiding the aliasing problem from selection of the nearest lattice point. Let the 1D texture have N samples. If i' is the continuous image index computed from the texture coordinate, then let $\ell_0 = \lfloor i' \rfloor$ (the floor of i') and $\ell_1 = \lceil i' \rceil$ (the ceiling of i'). The image values $\mathbf{C}_{\ell_0}$ and $\mathbf{C}_{\ell_1}$ are linearly interpolated to produce the pixel color

$$\mathbf{C} = (1 - \delta)\mathbf{C}_{\ell_0} + \delta\mathbf{C}_{\ell_1} \quad (2.96)$$

where $\delta = i' - \ell_0$.

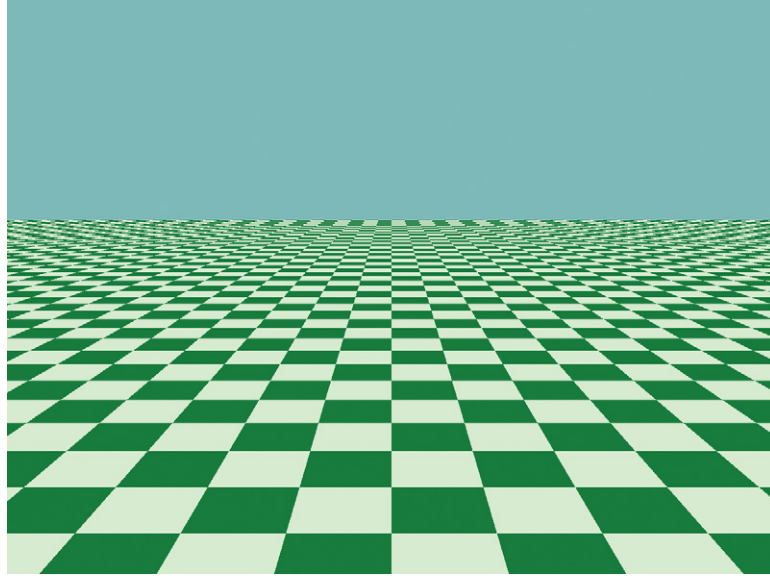


Figure 2.42 A texture image with nearest-neighbor filtering.

If the continuous image indices for a 2D image are (i', j') corresponding to a texture coordinate (s, t) , define ℓ_0 and ℓ_1 as for a 1D texture and define $m_0 = \lfloor j' \rfloor$ and $m_1 = \lceil j' \rceil$. The image values C_{ℓ_0, m_0} , C_{ℓ_1, m_0} , C_{ℓ_0, m_1} , and C_{ℓ_1, m_1} are bilinearly interpolated to produce the pixel color

$$C = (1 - \delta)((1 - \epsilon)C_{\ell_0, m_0} + \epsilon C_{\ell_0, m_1}) + \delta((1 - \epsilon)C_{\ell_1, m_0} + \epsilon C_{\ell_1, m_1}) \quad (2.97)$$

where $\delta = i' - \ell_0$ and $\epsilon = j' - m_0$.

If the continuous image indices for a 3D image are (i', j', k') corresponding to a texture coordinate (s, t, r) , define ℓ_0, ℓ_1, m_0 , and m_1 as was done for a 2D texture and define $n_0 = \lfloor k' \rfloor$ and $n_1 = \lceil k' \rceil$. The image values C_{ℓ_0, m_0, n_0} , C_{ℓ_1, m_0, n_0} , C_{ℓ_0, m_1, n_0} , C_{ℓ_1, m_1, n_0} , C_{ℓ_0, m_0, n_1} , C_{ℓ_1, m_0, n_1} , C_{ℓ_0, m_1, n_1} , and C_{ℓ_1, m_1, n_1} are trilinearly interpolated to produce the pixel color

$$\begin{aligned} C = & (1 - \delta)((1 - \epsilon)((1 - \phi)C_{\ell_0, m_0, n_0} + \phi C_{\ell_0, m_0, n_1}) \\ & + \epsilon((1 - \phi)C_{\ell_0, m_1, n_0} + \phi C_{\ell_0, m_1, n_1})) \\ & + \delta((1 - \epsilon)((1 - \phi)C_{\ell_1, m_0, n_0} + \phi C_{\ell_1, m_0, n_1}) \\ & + \epsilon((1 - \phi)C_{\ell_1, m_1, n_0} + \phi C_{\ell_1, m_1, n_1})) \end{aligned} \quad (2.98)$$

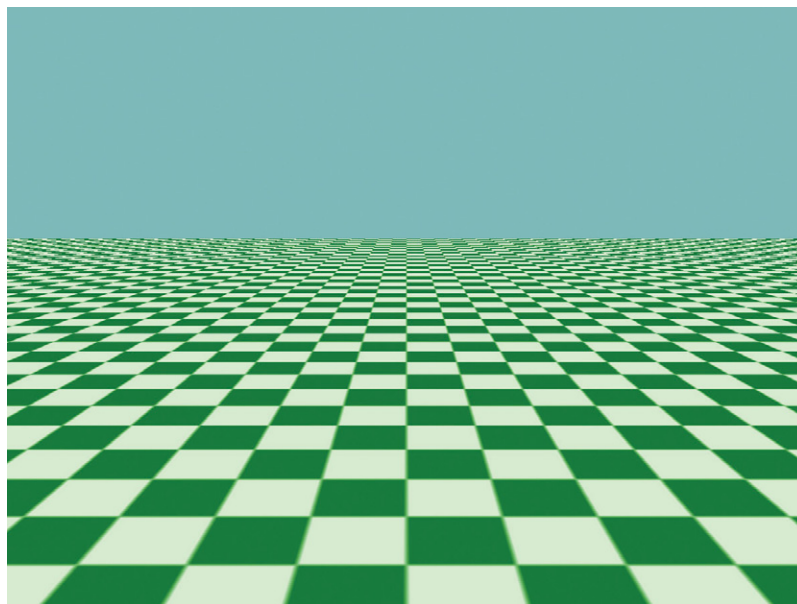


Figure 2.43 A texture image with bilinear filtering.

The texture image of Figure 2.42 was drawn using bilinear interpolation. The result is shown in Figure 2.43. Notice that the jaggedness of Figure 2.43 does not appear in the foreground of this figure, but there is jaggedness in the background of the figure. How to deal with this is the topic of *mipmapping*, which I discuss later in this section.

As promised, here is a comparison of the behavior of the texture coordinate modes at the boundary of an image. For illustration, the comparison is shown for a 1D image at its left boundary. The image samples are C_i for $0 \leq i \leq N - 1$ and the border color is B . Table 2.3 shows the behavior when nearest-neighbor filtering is used. Recall that rounding is applied so that numbers x with $x - \lfloor x \rfloor < 1/2$ are rounded to $\lfloor x \rfloor$ and numbers x with $x - \lfloor x \rfloor \geq 1/2$ are rounded to $\lceil x \rceil$.

To remind you of interval notation, $[a, b]$ is the set of x for which $a \leq x \leq b$. The interval $[a, b)$ has the constraint $a \leq x < b$; the interval $(a, b]$ has the constraint $a < x \leq b$; and the interval (a, b) has the constraint $a < x < b$. Notice the difference in behavior between clamp mode and clamp-to-border mode.

The more interesting behaviors, and ones that tend to catch your attention visually when building terrains from height fields, occur when you use linear filtering (bilinear for 2D images). Dark seams between terrain pages is an indication that you have the wrong clamping mode for the textures assigned to the pages. Table 2.4 shows

Table 2.3 Selected color for a continuous image index when nearest-neighbor filtering is used.

i Interval	Clamp	Clamp to Edge	Clamp to Border	Repeat	Mirrored Repeat
$(-3/2, -1/2)$	C_0	C_0	B	C_{N-1}	C_0
$[-1/2, 0)$	C_0	C_0	C_0	C_0	C_0

Table 2.4 Selected color for a continuous image index when linear filtering is used. The pairs of colors are those used in the linear interpolation. The weights applied to these depends on where in the interval i occurs.

i Interval	Clamp	Clamp to Edge	Clamp to Border	Repeat	Mirrored Repeat
$(-2, -1]$	B, C_0	C_0, C_0	B, B	C_{N-2}, C_{N-1}	C_1, C_0
$(-1, 0)$	B, C_0	C_0, C_0	B, C_0	C_{N-1}, C_0	C_0, C_0

the behavior when linear filtering is used. Notice the difference in behavior between clamp mode and clamp-to-border mode.

The Wild Magic software renderer implements texture sampling, including the conversion of texture coordinates to image coordinates and the filtering using nearest neighbors or linear interpolation. The file containing the conversion of texture coordinates is `Wm4SoftSampler.cpp`. The files containing the sampling and filtering are `Wm4SoftSampler1.cpp`, `Wm4SoftSampler2.cpp`, and `Wm4SoftSampler3.cpp`, where the final number in the file name is the dimension of the image.

One thing you should notice in the sampling code is that I require the image dimensions to be powers of two. For a software renderer, this is not a constraint you really need to have, but I have made it in order to illustrate rapid lookups in an image stored as a 1D array in memory. For a 2D image of dimensions $N \times M$ stored in row-major order, the image value at location (x, y) is stored in linear memory at the index $i = x + Ny$. Computing the index requires a multiplication, something that can be expensive to compute in hardware (when you have a lot of multiplications). If $N = 2^p$ for $p > 0$, the index calculation may be written instead as $i = x + (y \ll p)$, where the symbol $\ll$ denotes the shift-left operator. Shifting the bits of y to the left by p units is equivalent to multiplying by 2^p . Shifting is less expensive to compute in hardware than multiplication. Given the enormous number of texture image lookups during rendering, the time savings can be significant.

Mipmapping

Even bilinear filtering can have aliasing problems when a textured triangle is in the distance. As the distance from the eye point increases, the perceived frequency in the texture increases because the same range of texture coordinates is applied over the smaller set of pixels covered by the triangle. This produces a temporal aliasing of the textures on objects close to the far plane. A method for reducing the aliasing is *mipmapping* [Wil83]. The prefix *mip* is an acronym for the Latin *multum in parvo*, which means “many things in a small place.” The idea is that a pyramid of textures is built from the original by downsampling via averaging or blurring.

To illustrate downsampling by averaging, consider a 1D image with $N_0 = 2^{p_0}$ samples. A smaller image is constructed with half the samples. If $C_0(i)$ are the original image samples for $0 \leq i < 2^{p_0}$, the downsampled image samples are $C_1(i)$ for $0 \leq i < 2^{p_0-1}$. Specifically, they are averages:

$$C_1(i) = 0.5(C_0(2i) + C_0(2i + 1))$$

The downsampling is repeated on C_1 to obtain a smaller image C_2 . The process continues until the final image has one sample. The number of such images is $p_0 + 1$, including the original image. In the Wild Magic software renderer, 1D downsampling is computed by the function `SoftSampler::RecreateMipmap1` in the file `Wm4SoftSampler.cpp`. The pyramid construction is computed by the function `SoftSampler1::CreateMipmaps`.

Now consider a 2D image of size $2^{p_0} \times 2^{p_1}$. For the sake of argument, let $p_0 \geq p_1$. The downsampling occurs by averaging 2×2 blocks of pixels. Assuming $p_1 > 1$, the first downsampled image has size $2^{p_0-1} \times 2^{p_1-1}$. Using the same notation as before, the averaging is

$$C_1(i, j) = 0.25(C_0(2i, 2j) + C_0(2i + 1, 2j) + C_0(2i, 2j + 1) + C_0(2i + 1, 2j + 1))$$

The downsampling is repeated until you reach an image size where one of the bounds is 1. Since we have assumed $p_0 \geq p_1$, the image you reach has size $2^{p_0-p_1} \times 1$. If $p_0 > p_1$, you can use the downsampling algorithm that was applied to 1D images. However, you still need to represent the image as a 2D image because the lookups use 2D texture coordinates. There are $p_0 + 1$ images in the pyramid when $p_0 \geq p_1$. Naturally, an implementation must deal with both cases $p_0 \geq p_1$ and $p_0 < p_1$. In the Wild Magic software renderer, 2D downsampling is computed by the function `SoftSampler::RecreateMipmap2`. The pyramid construction is computed by the function `SoftSampler2::CreateMipmaps`.

Finally, consider a 3D image of size $2^{p_0} \times 2^{p_1} \times 2^{p_2}$. For the sake of argument, let $p_0 \geq p_1 \geq p_2$. The downsampling occurs by averaging $2 \times 2 \times 2$ cubes of voxels. Assuming $p_2 > 0$, the first downsampled image has size $2^{p_0-1} \times 2^{p_1-1} \times 2^{p_2-1}$. The averaging is

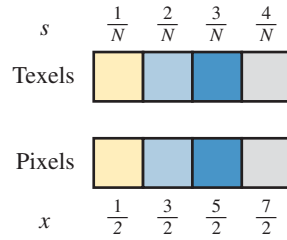


Figure 2.44 A one-to-one correspondence between texels of the texture image and pixels of the display window.

$$\begin{aligned}
 C_1(i, j, k) = & 0.125 (C_0(2i, 2j, 2k) + C_0(2i + 1, 2j, 2k) + C_0(2i, 2j + 1, 2k) \\
 & + C_0(2i + 1, 2j + 1, 2k + 1) + C_0(2i, 2j, 2k + 1) \\
 & + C_0(2i + 1, 2j, 2k + 1) + C_0(2i, 2j + 1, 2k + 1) \\
 & + C_0(2i + 1, 2j + 1, 2k + 1))
 \end{aligned}$$

The downsampling is repeated until you reach an image of size $2^{p_0-p_2} \times 2^{p_1-p_2} \times 1$. If $p_1 > p_2$, you can use the downsampling algorithm that was applied to 2D images, but representing these images as 3D (one of the bounds is always 1). If $p_0 > p_1 = p_2$, you can use the downsampling algorithm that was applied to 1D images, also representing these as 3D images. In the Wild Magic software renderer, 3D downsampling is computed by the function `SoftSampler::RecreateMipmap3`. The pyramid construction is computed by the function `SoftSampler3::CreateMipmaps`.

Downsampling the images to form a pyramid of images is half of the mipmapping process. The other half is to devise an algorithm for selecting which of the pyramid images to use when applying a texture to a pixel. For motivation, suppose that you have a texture image that is $N \times N$ and it is applied to a square mesh (two triangles) whose rendering covers a square portion of the window. Moreover, suppose that the covered window pixels are also of size $N \times N$. Essentially, you have one texel of the texture image assigned to one pixel of the window, as Figure 2.44 illustrates.

The texture coordinates shown in Figure 2.44 are computed during rasterization, based on the texture coordinates assigned to vertices of the square, and assigned to the pixels. Now suppose that the texture coordinates vary at twice the rate shown in Figure 2.44, but for the same set of pixels. Figure 2.45 illustrates this.

The pixel at $x = 1/2$ received the same color as in the previous figure, the one occurring at $s = 1/N$. However, the faster rate of change of the texture coordinate caused the pixel at $x = 3/2$ to receive the color for texture coordinate $s = 3/N$. The color at texture coordinate $s = 2/N$ was completely missed by the sampling. Imagine



Figure 2.45 The texture coordinates vary twice as fast as those of Figure 2.44.

yet a larger rate of change of the texture coordinates. Consecutive pixels are assigned texels that are quite far apart in the texture image.

The visual effect when the texture coordinates vary at a high rate over a block of consecutive pixels is shown in the background of Figures 2.42 and 2.43. These two figures are different, though, in the foreground. The nearest-neighbor filtering causes jaggedness because of the abrupt change from selecting a light blue texel for one pixel, and then selecting a blue texel for an adjacent pixel. The bilinear filtering removes the jaggedness, replacing an abrupt change from blue to light blue by a smooth linear ramp.

The bilinear filtering, however, cannot correct the problem in the background. Two adjacent pixels are assigned two nonadjacent and well-separated texels. The weighted averaging provided by bilinear filtering applies only to the two texel colors. It does not take into account any of the texel colors between those two. This is where mipmapping and the image pyramid come into play. Each downsampled image averages blocks of adjacent colors, so in a sense the lowest-resolution images in the pyramid contain color information from the highest-resolution images. Instead of bilinearly interpolating the original image for the distant pixels, one of the lower-resolution images in the pyramid is selected and the bilinear interpolation applied to it. The two texels used in the bilinear interpolation now have color information related to those texels that were skipped when only the original image was used. Figure 2.46 conveys the idea that a row of pixels close to the observer uses the original image, but a row of pixels farther away requires the downsampled image. Returning to our checkerboard example, the jaggedness in the background of Figure 2.43 is smoothed out, producing the image of Figure 2.47.

I have not stated precisely all the details involved in mipmapping. The previous example implies that I selected one of the images in the pyramid and used it. In fact, there are a few options for sampling the pyramid to obtain a final color for a pixel:

- Nearest-nearest filtering. The mipmap image nearest to the pixel is selected. In that image, the texel nearest to the pixel is selected and assigned to the pixel.

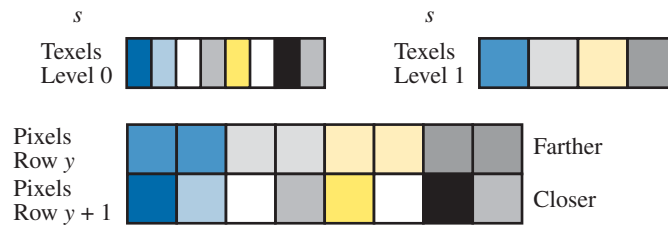


Figure 2.46 Two rows of pixels, the bottom row using the original image for texturing and the top row using a downsampled image obtained by averaging.

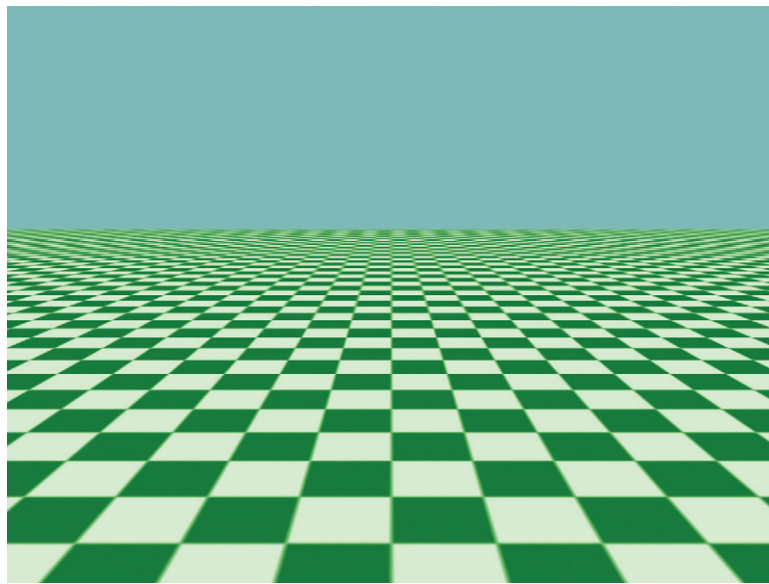


Figure 2.47 A texture image with linear-linear mipmapping filtering.

- Nearest-linear filtering. The two mipmap images that bound the pixel are selected. In each image, the texel nearest to the pixel is selected. The two texels are linearly interpolated and assigned to the pixel.
- Linear-nearest filtering. The mipmap image nearest to the pixel is selected. In that image, bilinear interpolation of the texels is used to produce the pixel value.

- Linear-linear filtering. The two mipmap images that bound the pixel are selected. In each image, bilinear interpolation of the texels is used to generate two colors. Those colors are linearly interpolated to produce the pixel value. This is sometimes called *trilinear filtering*.

Note that for each filter, the first name refers to the interpolation type within an image. The second name refers to the interpolation type across two images.

Of course, it is not yet clear what it means to select the image nearest to the pixel or to select the two images that bound the pixel. The motivating example discussed previously made it clear that the rate of change of the texture coordinates as you visit adjacent pixels is the key factor here. Once we quantify such a rate of change, we may use it for selecting the images to use in the image pyramid.

The relationship between a window point (x, y) and *any* vertex attribute A is constructed as follows, and uses the material discussed in Section 2.2.4. A triangle in world space has vertices $\mathbf{Q}_i$ for $0 \leq i \leq 2$. The triangle is parameterized by

$$\mathbf{Q}(s, t) = \mathbf{Q}_0 + s(\mathbf{Q}_1 - \mathbf{Q}_0) + t(\mathbf{Q}_2 - \mathbf{Q}_0)$$

where $s \geq 0, t \geq 0$, and $s + t \leq 1$. The s and t here are not to be confused with texture coordinates. The only vertex attribute of interest here has been named A , so you can think of A as a component of the texture coordinates. The corresponding window points are $\mathbf{P}_i = (x_i, y_i)$. In Section 2.2.4, we saw that perspective projection produces the representation for the projected triangle,

$$(x, y) = (x_0, y_0) + \bar{s}(x_1 - x_0, y_1 - y_0) + \bar{t}(x_2 - x_0, y_2 - y_0) \quad (2.99)$$

Equations (2.45) and (2.46) are the mappings between (s, t) and $(\bar{s}, \bar{t})$.

Let A_i be the vertex attributes for the triangle. The attributes for the entire triangle are

$$A = A_0 + s(A_1 - A_0) + t(A_2 - A_0) \quad (2.100)$$

We may solve Equation (2.99) to obtain $(\bar{s}, \bar{t})$ as a function of (x, y) , and then substitute this into Equation (2.46) to obtain (s, t) as a function of (x, y) . This result is substituted into Equation (2.100) to obtain an equation of the form

$$A(x, y) = \frac{ax + by + c}{dx + ey + f} \quad (2.101)$$

This is called a *fractional linear transformation*, an expression that appears regularly in projective geometry. The coefficients a through f depend on the (x_i, y_i) and A_i values.

Equation (2.101) quantifies how the attribute A varies with respect to the pixel coordinates. In particular, the rates of change are partial derivatives

$$\frac{\partial A}{\partial x} = \frac{(ae - bd)y + (af - cd)}{(dx + ey + f)^2}, \quad \frac{\partial A}{\partial y} = \frac{(bd - ae)y + (bf - ce)}{(dx + ey + f)^2} \quad (2.102)$$

From these rates of change, we want a *summary statistic*—a single number that allows us to select the images we want to use in the mipmap image pyramid.

For a 1D image, the rate of change of the texture coordinate s may be summarized by a statistic, but we need to normalize our comparison between texture coordinates and image coordinates. Specifically, let $u(x, y) = Ns(x, y)$, where the image has $N = 2^n$ samples. Choose

$$d = \log_2 \sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2} \quad (2.103)$$

The logarithm is base 2 and is applied because the downsampling creates $\log_2(N) = n$ images. If $d = 0$, effectively u varies at the same rate as the pixels are varying, so you would select the original image in the pyramid. For smaller values of d , you still choose the original image. For a value $d = 1$, u varies about twice as fast as the pixel location, so you would choose the image that was downsampled from the original. As d increases, you choose lower-resolution images in the pyramid. For the purpose of selecting two images to bound a pixel, calculate d and choose the floor and ceiling as the indices into the image pyramid.

For a 2D image, there are a few ways to summarize the rate of change of the texture coordinate (s, t) . As in the 1D case, we need to normalize the texture coordinates to be able to compare them to image coordinates. Define $u(x, y) = 2^n s(x, y)$ and $v(x, y) = 2^m t(x, y)$ where the image has size $2^n \times 2^m$. The first-order partial derivatives may be written as the entries of a matrix, which is called the *derivative matrix*, or *Jacobian matrix*, of the transformation from (x, y) to (u, v) ,

$$J = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial v}{\partial x} \\ \frac{\partial u}{\partial y} & \frac{\partial v}{\partial y} \end{bmatrix}$$

From linear algebra, it is known that the absolute value of the determinant of J is a measure of how the infinitesimal area at (x, y) is magnified to an infinitesimal area at (u, v) . The mipmap selection variable is set to

$$d = \log_2 |\det(J)| = \log_2 \left| \frac{\partial u}{\partial x} \frac{\partial v}{\partial y} - \frac{\partial u}{\partial y} \frac{\partial v}{\partial x} \right| \quad (2.104)$$

and is an approximate measure of how many texels are required to cover the pixel (x, y) . If $d \leq 1$, then the original image is used for texturing the pixel. As d increases, you use lower-resolution images in the image pyramid.

As a summary statistic, d measures a change in infinitesimal area in an isotropic manner, not caring about how the texture coordinate changes in any particular direction. This can be a problem visually. Consider, for example, texture coordinates (s, t) for which s varies at twice the rate x does but does not vary in y , and for which

t varies at half the rate y does but does not vary in x . The Jacobian matrix is

$$J = \begin{bmatrix} 2 & 0 \\ 0 & 1/2 \end{bmatrix}$$

The selection is $d = \log_2 |\det(J)| = 0$, indicating we should use the original image for texturing. However, the jaggedness will show up in the x -direction but not the y -direction. The effect can be very pronounced; mathematically, $\partial u/\partial x$ can be 1000 and $\partial v/\partial y$ can be $1/1000$, and still $d = 0$.

One alternative is to use a different summary statistic. For example, you could use the Euclidean norm of J and choose the mipmap selection

$$d = \log_2 \|J\| = \log_2 \sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2} \quad (2.105)$$

This generalizes the idea from one dimension, where the length of the gradient vector was used for the summary statistic. In the example where s varied twice the rate of x and t varied half the rate of y , we have $\log_2 \|J\| = \log_2 \sqrt{4.25} \doteq 1.04$, so you would select the first downsampled image ($d = 1$) and not the original image. Another alternative is to choose the mipmap selection to be

$$d = \log_2 \max \left\{ \sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2} \right\} \quad (2.106)$$

OpenGL uses this selection mechanism. Even these selections cannot handle extreme cases when one of the texture components varies greatly and the other does not. The problem is not so much the summary statistic as it is the downsampling by averaging. The averaging itself is isotropic, not caring about texture image variation in some preferred direction. The end result is that overblurring occurs in the direction of least variation of the texture components.

An attempt to reduce the isotropic effects is to use *ripmaps* [MB05]. The averaging process to obtain a sequence of blurred images is applied independently in each dimension. The lookup process now involves two parameters, one related to the length of the gradient of s and one related to the length of the gradient of v , and then using the most appropriate one for the texel lookup. Yet even this can have problems because the downsampling is still performed in the directions of the image axes.

The most common method to handle the mipmapping these days is *anisotropic filtering*. The idea gets right to the heart of the matter—determining those texels that actually should cover the image and sampling accordingly. For a 2D image with texture coordinates (s, t) , the mapping from (x, y) to the scaled coordinates (u, v) comes from Equation (2.101), but as applied to two different attributes,

$$u = \frac{a_0x + b_0y + c_0}{dx + ey + f}, \quad v = \frac{a_1x + b_1y + c_1}{dx + ey + f}$$

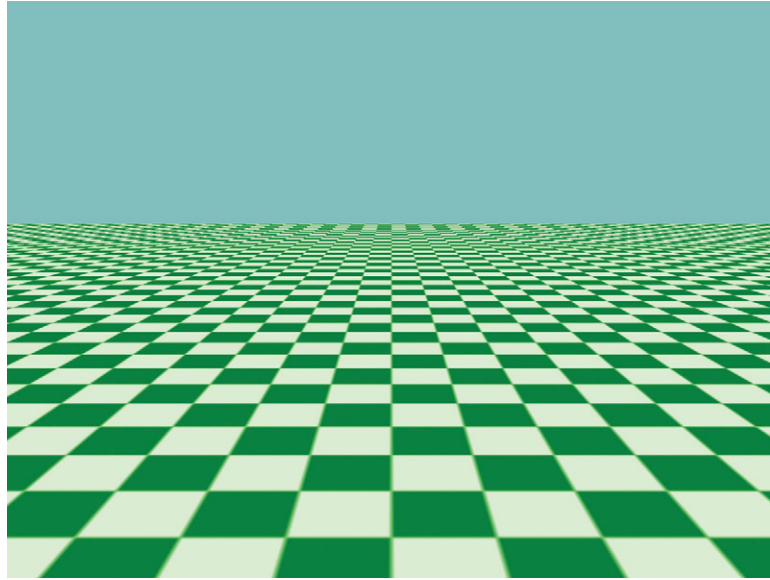


Figure 2.48 A texture image with anisotropic filtering using 16 samples per pixel lookup.

This pair of equations maps convex quadrilaterals to convex quadrilaterals, a basic property of fractional linear transformations and perspective projection. A pixel square, therefore, maps to a convex quadrilateral in the texture image. This quadrilateral may be sampled multiple times and combined to produce the pixel color. The way that graphics hardware tends to sample is by constructing a line segment passing through the middle of the quadrilateral—in the long direction—and then sampling along that segment. The line segment is called the *line of anisotropy*. The longer the quadrilateral is in this direction compared to length in the other direction, the more samples you choose. Figure 2.48 shows the same checkerboard drawn previously with nearest-neighbor filtering, bilinear filtering, and isotropic trilinear filtering. The current figure uses anisotropic filtering with 16 samples chosen along the line of anisotropy.

The large-scale blurring that occurred with trilinear filtering has been replaced with some sharper detail, but the symmetry of the checkerboard still appears to create problems. Using two, four, or eight samples changed where the sharpness occurred, but the results were still not quite pleasing. Well, that's checkerboards for you. For a realistic environment, the anisotropic filtering works very well.

OpenGL has an extension for anisotropic filtering,

```
GL_EXT_texture_filter_anisotropic
```

If u_x , u_y , v_x , and v_y denote the partial derivatives of the scaled coordinates u and v , define $L_x = |(u_x, v_x)|$, $L_y = |(u_y, v_y)|$, $L_{\min} = \min\{L_x, L_y\}$, and $L_{\max} = \max\{L_x, L_y\}$. The number of anisotropic samples computed is

$$n = \min \left\{ \left\lceil \frac{L_{\max}}{L_{\min}} \right\rceil, a_{\max} \right\}$$

where the maximum anisotropic value $a_{\max}$ is specified by calling

```
float max_amax;
glFloatfv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &max_amax);
float amax = <your desired maximum, between 1 and max_amax>;
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, amax);
```

The mipmap selection is

$$d = \log_2 \left(\frac{L_{\max}}{n} \right)$$

where n is the number of anisotropic samples.

EXERCISE 2.16

The Wild Magic software renderer implements isotropic mipmapping using Equation (2.106) for the mipmap image selection. Modify the renderer to support anisotropic mipmapping. ■

Multitexture

The number of texture images associated with a triangle need not always be one. Multiple textures, or *multitextures*, allow for a lot of special effects that enhance the realism of the rendered scene. For example, multitextures can add variations in lighting to textures on the walls in a room. This is a form of *static multitexture*—the secondary texture corresponding to the lighting is combined with the primary texture corresponding to the walls in a view-independent manner. Combining such textures is a way to add visual variation in a scene without an exponential growth in texture memory usage. N primary textures and M secondary textures can be combined in NM ways, but only $N + M$ textures are required in memory as opposed to storing NM textures. Moreover, an artist can generate the smaller number of textures in less time.

Here is another example: A character moves along a textured floor in a scene with a light and casts a shadow on the floor. The shadow can be dynamically computed as a texture and is applied to the floor triangles. This is a form of *dynamic multitexture*—the secondary texture is generated on the fly. The triangles on which the shadow is cast must be selected by the application, and the corresponding texture coordinates must also be computed on the fly. In either case, you must decide how the textures

are to be combined to produce a single color for each pixel. Generally, this is called *blending* and is easy enough to implement with shader programming.

2.6.4 TRANSPARENCY, OPACITY, AND BLENDING

Given two RGBA colors, one called the *source color* and one called the *destination color*, the term *alpha blending* refers to the general process of combining the source and destination into yet another RGBA color. The source color is (r_s, g_s, b_s, a_s) , and the destination color is (r_d, g_d, b_d, a_d) . The blended result is the final color (r_f, g_f, b_f, a_f) . All color channel values in this discussion are assumed to be in the interval $[0, 1]$. Material colors and texture image colors both can have alpha channels.

The classical method for blending is to use the alpha channel as an opacity factor. If the alpha value is 1, the color is completely opaque. If the alpha value is 0, the color is completely transparent. If the alpha value is strictly between 0 and 1, the colors are semitransparent. The formula for the blend of only the RGB channels is

$$\begin{aligned}(r_f, g_f, b_f) &= (1 - a_s)(r_s, g_s, b_s) + a_s(r_d, g_d, b_d) \\ &= ((1 - a_s)r_s + a_sr_d, (1 - a_s)g_s + a_sg_d, (1 - a_s)b_s + a_sb_d)\end{aligned}$$

The algebraic operations are performed component by component. The assumption is that you have already drawn the destination color into the frame buffer; that is, the frame buffer becomes the destination. The next color you draw is the source. The alpha value of the source color is used to blend the source color with the current contents of the frame buffer.

It is also possible to draw the destination color into an offscreen buffer, blend the source color with it, and then use the offscreen buffer for blending with the current contents of the frame buffer. In this sense, we also want to keep track of the alpha value in the offscreen buffer. We need a blending equation for the alpha values themselves. Using the same operations as for the RGB channels, your choice will be

$$a_f = (1 - a_s)a_s + a_sa_d$$

Combining the four channels into a single equation, the classic alpha blending equation is

$$\begin{aligned}(r_f, g_f, b_f, a_f) &= ((1 - a_s)r_s + a_sr_d, (1 - a_s)g_s + a_sg_d, \\ &\quad (1 - a_s)b_s + a_sb_d, (1 - a_s)a_s + a_sa_d)\end{aligned}\tag{2.107}$$

If the final colors become the destination for another blending operation, then

$$(r_d, g_d, b_d, a_d) = (r_f, g_f, b_f, a_f)$$

sets the destination to the previous blending results.

Table 2.5 Possible source blending coefficients.

<i>Enumerated Value</i>	$(\sigma_r, \sigma_g, \sigma_b, \sigma_a)$
SBF_ZERO	(0, 0, 0, 0)
SBF_ONE	(1, 1, 1, 1)
SBF_DST_COLOR	(r_d, g_d, b_d, a_d)
SBF_ONE_MINUS_DST_COLOR	$(1 - r_d, 1 - g_d, 1 - b_d, 1 - a_d)$
SBF_SRC_ALPHA	(a_s, a_s, a_s, a_s)
SBF_ONE_MINUS_SRC_ALPHA	$(1 - a_s, 1 - a_s, 1 - a_s, 1 - a_s)$
SBF_DST_ALPHA	(a_d, a_d, a_d, a_d)
SBF_ONE_MINUS_DST_ALPHA	$(1 - a_d, 1 - a_d, 1 - a_d, 1 - a_d)$
SBF_SRC_ALPHA_SATURATE	$(\sigma, \sigma, \sigma, 1), \sigma = \min\{a_s, 1 - a_d\}$
SBF_CONSTANT_COLOR	(r_c, g_c, b_c, a_c)
SBF_ONE_MINUS_CONSTANT_COLOR	$(1 - r_c, 1 - g_c, 1 - b_c, 1 - a_c)$
SBF_CONSTANT_ALPHA	(a_c, a_c, a_c, a_c)
SBF_ONE_MINUS_CONSTANT_ALPHA	$(1 - a_c, 1 - a_c, 1 - a_c, 1 - a_c)$

Graphics APIs support more general combinations of colors, whether the colors come from vertex attributes or texture images. The general equation is

$$(r_f, g_f, b_f, a_f) = (\sigma_r r_s + \delta_r r_d, \sigma_g g_s + \delta_g g_d, \sigma_b b_s + \delta_b b_d, \sigma_a a_s + \delta_a a_d) \quad (2.108)$$

The blending coefficients are σ_i and δ_i , where the subscripts denote the color channels they affect. The coefficients are assumed to be in the interval $[0, 1]$. The σ_i are specified indirectly through enumerations referred to as the *source blending functions*. The δ_i are also specified indirectly through enumerations referred to as the *destination blending functions*. I will use the prefix SBF for the source blending functions. Table 2.5 lists the possibilities for the source blending coefficients. The constant color, (r_c, g_c, b_c, a_c) , is specified independently of material or texture colors. The prefix DBF is used for the destination blending functions. Table 2.6 lists the possibilities for the destination blending coefficients.

Table 2.5 has DST_COLOR, ONE_MINUS_DST_COLOR, and SRC_ALPHA_SATURATE, but Table 2.6 does not. Table 2.6 has SRC_COLOR and ONE_MINUS_SRC_COLOR, but Table 2.5 does not. The classic alpha blending equation (2.107) occurs when the source blending function is SBF_SRC_ALPHA and the destination blending function is DBF_ONE_MINUS_SRC_ALPHA.

Here is an interesting example to compare some blending modes. Two textures may be multiplied together. The source blending function is SBF_DST_COLOR and the

Table 2.6 Possible destination blending coefficients.

<i>Enumerated Value</i>	$(\delta_r, \delta_g, \delta_b, \delta_a)$
DBF_ZERO	(0, 0, 0, 0)
DBF_ONE	(1, 1, 1, 1)
DBF_SRC_COLOR	(r_s, g_s, b_s, a_s)
DBF_ONE_MINUS_SRC_COLOR	$(1 - r_s, 1 - g_s, 1 - b_s, 1 - a_s)$
DBF_SRC_ALPHA	(a_s, a_s, a_s, a_s)
DBF_ONE_MINUS_SRC_ALPHA	$(1 - a_s, 1 - a_s, 1 - a_s, 1 - a_s)$
DBF_DST_ALPHA	(a_d, a_d, a_d, a_d)
DBF_ONE_MINUS_DST_ALPHA	$(1 - a_d, 1 - a_d, 1 - a_d, 1 - a_d)$
DBF_CONSTANT_COLOR	(r_c, g_c, b_c, a_c)
DBF_ONE_MINUS_CONSTANT_COLOR	$(1 - r_c, 1 - g_c, 1 - b_c, 1 - a_c)$
DBF_CONSTANT_ALPHA	(a_c, a_c, a_c, a_c)
DBF_ONE_MINUS_CONSTANT_ALPHA	$(1 - a_c, 1 - a_c, 1 - a_c, 1 - a_c)$

blending function is DBF_ZERO. The blending equation is

$$(r_f, g_f, b_f, a_f) = (r_d r_s, g_d g_s, b_d b_s, a_d a_s) \quad (2.109)$$

Multiplication of color channels in $[0, 1]$ results in values in $[0, 1]$, so no clamping is needed. This type of blending is used for *dark maps*, where the source texture represents a simulated light source. The term *dark* is used because the multiplication can lower the intensity of the final color compared to the destination color. Two textures may be added together. The source blending function is SBF_ONE and the blending function is DBF_ONE. The blending equation is

$$(r_f, g_f, b_f, a_f) = (r_s + r_d, g_s + g_d, b_s + b_d, a_s + a_d) \quad (2.110)$$

Addition of color channels may lead to values larger than 1, so the final color channels are clamped to the interval $[0, 1]$. This type of blending is used for *light maps*, where the source texture also represents a simulated light source. The final colors tend to increase in intensity and appear to be oversaturated.

An alternative to additive blending is *soft addition*. The formula is

$$\begin{aligned} (r_f, g_f, b_f, a_f) = & ((1 - r_d)r_s, (1 - g_d)g_s, (1 - b_d)b_s, (1 - a_d)a_s) \\ & + (r_d, g_d, b_d, a_d) \end{aligned} \quad (2.111)$$

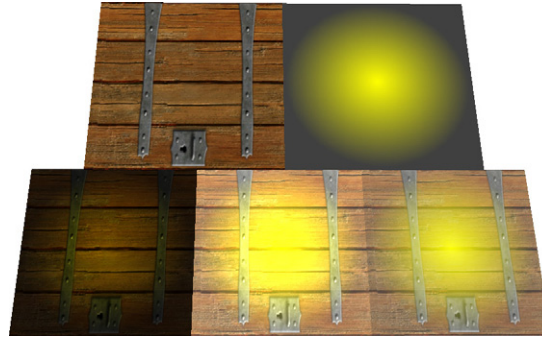


Figure 2.49 Upper left: A primary texture is a wooden image. Upper right: A secondary texture to blend with the primary texture. Lower left: A dark map using multiplicative blending. Lower middle: A light map using additive blending. Lower right: A light map using soft additive blending.

The source blend function is `SBF_ONE_MINUS_DST_COLOR` and the destination blend function is `DBF_ONE`. This approach does not lead to oversaturation. The idea is that you start with the destination color and add a fraction of the source color to it. If the destination color is bright (values near 1), then the source blend coefficients are small, so the source color will not cause the result to wash out. Similarly, if the destination color is dark (values near 0), the destination color has little contribution to the result, and the source blend coefficients are large, so the source color dominates and the final result is a brightening of dark regions. Figure 2.49 shows two textures to be blended together, and the results of multiplicative blending, additive blending, and soft additive blending are shown.

The concept of blending also encapsulates what is referred to as *alpha testing*. The idea is that an RGBA source color will only be combined with the RGBA destination color as long as the source alpha value compares favorably with a specified *reference* value. Pseudocode for alpha testing is

```
source = (Rs,Gs,Bs,As);
destination = (Rd,Gd,Bd,Ad);
reference = Aref;
if (ComparesFavorably(As,Aref))
{
    result = BlendTogether(source,destination);
}
```

The `ComparesFavorably(x,y)` function is a standard comparison between two numbers: $x < y$, $x \leq y$, $x > y$, $x \geq y$, $x = y$, or $x \neq y$. Two additional functions are allowed: `always` or `never`. In the former, the blending always occurs. This is the default behavior of an alpha blending system. In the latter, blending never occurs.

In order to correctly draw a list of objects, some semitransparent (alpha values smaller than 1), the rule is to draw your opaque objects first, and then draw your semitransparent objects sorted from back to front in the view direction. Game programmers are always willing to take a shortcut to obtain a faster system, or to avoid having to implement some complicated system, and hope that the consequences are not visually distracting. In this example, the shortcut is to skip the sorting step and use alpha testing. The list of objects is rendered twice, and on both passes, depth buffering is enabled in order to correctly sort the objects (on a per-pixel basis in window space). Also on both passes, alpha testing is enabled. On the first pass, the test function is set to allow blending for any colors with an alpha value equal to 1; that is, the opaque objects are drawn, but the semitransparent objects are not. On the second pass, the test function is set to allow blending for any colors with an alpha value not equal to 1. This time the semitransparent objects are drawn, but the opaque objects are not.

As stated, this system is not quite right (ignoring the back-to-front sorting issue). Depth buffering is enabled, but recall that you have the capability to control whether reading or writing occurs. The opaque objects are drawn in the first pass. The depth buffering uses both reading and writing to guarantee that the final result is rendered correctly. Before drawing a pixel in the frame buffer, the depth buffer is read at the corresponding location. If the incoming depth passes the depth test, then the pixel is drawn in the frame buffer. Consequently, the depth buffer must be written to update the new depth for this pixel. If the incoming depth does not pass the test, the pixel is not drawn, and the depth buffer is not updated. Semitransparent objects are drawn in the second pass. These objects were not sorted from back to front. It is possible that two semitransparent objects are drawn front to back; that is, the first drawn object is closer to the observer than the second drawn object. You can see through the first drawn object because it is semitransparent, so you expect to see the second drawn object immediately behind it. To guarantee that this happens, you have to disable depth buffer writes on the second pass. Consider what would happen if you did not do this. The first object is drawn, and the depth buffer is written with the depths corresponding to that object. When you try to draw the second object, its depths are larger than those of the first object, so the depth test fails and the second object is not drawn, even though it should be visible through the first. Disabling the depth buffer writing will prevent this error. Pseudocode to implement the process is

```
// initialization
ObjectList objects = <objects to draw, some opaque, some transparent>;
EnableDepthRead();
SetDepthCompare(less_than_or_equal);
```



```

EnableAlphaBlending(SBF_SRC_ALPHA,DBF_ONE_MINUS_SRC_ALPHA);
EnableAlphaTesting(1); // alpha reference = 1

// first pass
SetAlphaTestCompare(equal);
EnableDepthWrite();
Draw(objects);

// second pass
SetAlphaTestCompare(not_equal);
DisableDepthWrite();
Draw(objects);

```

Another application for alpha testing is for drawing objects that have textures whose alpha values are either 0 or 1. The idea is that the texture in some sense defines what the object is. A classic example involves applying a decal to an object. The decal geometry is a rectangle that has a texture associated with it. The texture image has an artistically drawn object that does not cover all the image pixels. The pixels not covered are “see-through”; that is, if the decal is drawn on top of another object, you see what the artist has drawn in the image, but you see the other object elsewhere. To accomplish this, the alpha values of the image are set to 1 wherever the artist has drawn, but to 0 everywhere else. When drawing such objects, the alpha reference value is set to 0.5 (it just needs to be different from 0 and 1), and the test function is set to be “greater than.” When the decal texture is drawn on the object, only the portion with alpha values equal to 1 (greater than 0.5) is drawn. The portion with alpha values equal to 0 (not greater than 0.5) is not drawn. Because they are not drawn, the depth buffer is not affected, so you do not have to use the two-pass technique discussed in the previous example.

2.6.5 FOG

The addition of fog to an image adds to the realism of the image and also helps to hide clipping artifacts at the far plane. Without fog, as the eye point moves away from an object, the object approaches the far plane and is noticeably clipped when the far plane intersects it. With fog, if the fog density increases with distance from the eye point, the effect is to provide a depth cue for objects in the distance. And if the fog density increases to full opacity at the far plane, clipping is substantially hidden and the objects disappear in a more natural fashion. If C_{fog} is the designated fog color, C_{pixel} is the current pixel color, and $\phi \in [0, 1]$ is the fog factor and is proportional to distance from the eye point, then the final color C_{final} is

$$C_{\text{final}} = (1 - \phi)C_{\text{pixel}} + \phi C_{\text{fog}}$$

There are a variety of ways to generate the fog factor. The standard way, called *linear fog*, is based on the z -value (or w -value) of the pixel to be fogged. Moreover, the fog can be applied to a subset $[z_0, z_1] \subseteq [d_{\min}, d_{\max}]$ of the view frustum. The linear fog factor is

$$\phi = \begin{cases} 0, & z < z_0 \\ \frac{z-z_0}{z_1-z_0}, & z \in [z_0, z_1] \\ 1, & z > z_1 \end{cases}$$

Since the z -values or w -values are computed by the renderer for other purposes, linear fog is relatively inexpensive to compute compared to other fog methods.

Exponential fog is obtained by allowing the fog to increase exponentially with the z -value of the pixel to be fogged,

$$\phi = \exp(\lambda z)$$

where $\lambda > 0$ is a parameter that controls the rate of increase with respect to z .

Range-based fog assigns the fog factor based on the distance r from eye point to pixel. A subset of radial values $[r_0, r_1]$ can be used, just as in linear fogging:

$$\phi = \begin{cases} 0, & r < r_0 \\ \frac{r-r_0}{r_1-r_0}, & r \in [r_0, r_1] \\ 1, & r > r_1 \end{cases}$$

This type of fog is more expensive to compute than linear fog since the distance must be calculated for each rendered pixel.

Another possibility for fog is to assign a factor per triangle vertex and let the rasterizer interpolate the factors over the entire triangle. This effect is used in volumetric fog; see Section 20.14 for a sample application.

2.6.6 AND MANY MORE

With the advent of shader programming, computer graphics practitioners and researchers have been quite creative in generating programs to obtain special effects. Some of these have a geometric flavor to them in that a vertex has attributes, including a vertex normal and two tangent vectors perpendicular to the normal and to each other. These attributes may be interpolated by the rasterizer just like vertex colors and texture coordinates are interpolated. In fact, vertex normals and tangents may be stored as texture coordinates (s, t, r) , even though they are not used for texture lookups. The vertex shader might even generate such attributes just so that they are passed back to the rasterizer, interpolated by the rasterizer, and then sent to the pixel

shader. This shader interprets the incoming texture coordinates as normals and tangents, and then uses them for per-pixel effects that take advantage of the coordinate frame defined by the normal and tangents.

What you decide to store in vertex attributes is limitless. You will find very creative shaders in books such as [Eng02, Eng03, Eng04, Eng06, Fer04, Pha05]. Some of the sample applications in Chapter 20 make use of assigning data to vertex attributes (in the color or texture channels) or generating the data itself.

2.6.7 RASTERIZING ATTRIBUTES

The rasterizer interpolates vertex attributes in two places when processing triangles. The first place is in the edge buffer construction, which was discussed in Section 2.5.4. This code has the job of interpolating the vertex attributes along edges connecting the triangle vertices. Once the edge buffers have been initialized with these attributes, the scan line rasterization occurs. This process itself must interpolate vertex attributes, namely, the two sets of attributes occurring at the endpoints of the scan line that were initialized by the edge buffer construction. The interpolation also occurs for wireframe mode when drawing edges.

Let w_0 and w_1 be the clip-space w -values for the endpoints of the span of pixels to be interpolated. Let the indices for the pixels be i_0 and i_1 and let i be the variable index to vary between these. For a scan line, i is the x -index of the pixel and the span of pixels is for a constant y . For edge buffering, i is the y -index of the pixel. Edge buffering guarantees that the y -values increment over consecutive rows. The x -value for each scan line is what the edge buffer computes. For edge rasterization, the index is x or y depending on the slope of the edge—the usual decision made when applying Bresenham’s line-drawing algorithm.

Let A_0 and A_1 be one channel of a vertex attribute (a color, a component of a normal, a component of a texture coordinate) at the endpoints of the span of pixels. An entire attribute array is processed by the interpolation. The arrays at the endpoints include (r_0, u_0, d_0, w_0) and (r_1, u_1, d_1, w_1) , where the clip-space tuples have components r , u , and d , which are from the camera coordinate system. These attributes are followed by vertex colors or normals or texture coordinates (if any). Thus, the interpolator computes (r, u) positions, depth d , and w -values, but all in clip coordinates. The linear interpolation of the attributes is

$$A(i) = \frac{(A_1 - A_0)i + (A_0i_1 - A_1i_0)}{i_1 - i_0} \quad (2.112)$$

where $i_0 < i < i_1$. The perspective interpolation is

$$A(i) = \frac{(A_1w_0 - A_0w_1)i + (A_0w_1i_1 - A_1w_0i_0)}{(w_0 - w_1)i + (w_1y_1 - w_0y_0)} \quad (2.113)$$

You have the choice of using linear interpolation or perspective interpolation. When the camera model uses perspective projection, it is appropriate to interpolate according to that projection. Orthographic camera models may use linear interpolation, but perspective projection produces the same results—the w -components are always 1 and the perspective divide changes nothing. The Wild Magic software renderer detects when the w -components are the same for the endpoints of a span of pixels to be interpolated and switches to linear interpolation. The function that performs the interpolation is `SoftRenderer::PerspectiveInterpolate` and is called during edge rasterization, `SoftRenderer::RasterizeEdge`, and during triangle rasterization, `SoftRenderer::RasterizeTriangle`. All three functions are in the file `Wm4SoftDrawElements.cpp`. It is also called during edge buffer construction, in function `SoftRenderer::ComputeEdgeBuffer` in the file `Wm4SoftEdgeBuffers.cpp`.

2.7 ISSUES OF SOFTWARE, HARDWARE, AND APIS

Although a graphics system may be designed abstractly and implemented in software, graphics hardware can provide many services that you build on top of. When doing so, there are issues you must think about and trade-offs to consider. I discuss some of these briefly in this section.

2.7.1 A GENERAL DISCUSSION

This chapter has described some relevant issues in building a renderer without regard to whether the work is done by a general-purpose CPU, in part by a hardware-accelerated graphics card, or totally by specialized graphics hardware. Independent of software or hardware, the rendering pipeline was also described without regard to integration with existing software that provides an application programmer interface (API). The reality of building a real-time computer graphics engine requires an understanding of which platforms are to be supported and which other existing systems can be used rather than implemented from scratch.

Graphics APIs such as Direct3D and OpenGL for consumer graphics accelerators can be viewed as providing a boundary between the scene graph management and the rendering system. Both are fairly high-level rendering APIs, and both attempt to hide the underlying hardware to allow an application to be portable across multiple hardware cards. Heated debates arise in the computer graphics and games newsgroups about whether Direct3D or OpenGL is the “best” system to build on. This is an unanswerable question—and in fact is not the question to ask. Each system has its advantages and disadvantages. As with most of computer science, the issue is more about understanding the trade-offs between using one system or another. In my opinion, OpenGL is clearly superior with respect to portability simply by its design. An application can be written to run on a high-end Silicon Graphics, Inc. (SGI) machine or on a consumer machine such as a PC or Macintosh. Direct3D was intended

only to provide portability among cards in a PC, and now in the Xbox and Xbox 360. On the other hand, OpenGL insists on handling many details that an application might like to control but cannot. Direct3D provides much more fine-grained control over the rendering process. Both APIs are constantly evolving based on what the end programmers want, but evolution takes time. Moreover, the consumer hardware cards are evolving at a fast enough rate that the drivers that ship with them are sometimes buggy but are not always corrected, because the next-generation card is almost ready to ship. This requires patching the layer on top of the APIs with workarounds for specific cards. Evolution is good, but fast evolution is painful, especially for a company producing a commercial product that runs on top of those cards and drivers. As hardware evolves and begins doing the higher-level work that the scene graph management system has been doing, the APIs should become easier to work with. However, there will always be work necessary on the scene graph side to feed data through the API. Direct3D and OpenGL are graphics systems. They are not systems to support complex animations, collision detection, physics, AI, or any of the other systems you find in a game engine. I believe most developers have found that you build the graphics system you need regardless of the underlying graphics API, and then you focus on other important aspects of the game.

Another part of the evolution of graphics on a consumer machine involves the CPUs themselves. Both Intel's Pentium chipsets and Advanced Micro Devices, Inc. (AMD) chipsets have evolved to include instructions to support a small amount of parallelism (SIMD: single instruction, multiple data) and to provide for faster operations such as inverse square roots (for normalizing vectors). To make the most of the new instructions, the registers of the CPUs must be loaded quickly. This requires having your data packaged in a suitable format. If your data is not formatted correctly, you must repackage to feed the registers, which invariably offsets most of the speedup for using SIMD. Again, portability among platforms becomes a significant issue simply because of data formats. The new CPUs also tend to have data alignment requirements that are not necessarily guaranteed by current-generation compilers, so either a memory manager must be written to handle the alignment or the chip companies must supply a compiler. In fact, current compilers have to catch up and provide automatic support for the new machine instructions, so it is essential to have additional compiler support from the chip companies. Game consoles have similar issues, most wanting 16-byte alignment to support 4-tuple operations. If the compilers do not automatically pack to 16 bytes, you must do so with your data structures.

Finally, one of the most important low-level aspects of building a renderer is cache coherence. Experience has shown that even with the best-designed high-level algorithms, the performance can be significantly reduced if the data is organized in such a way as to cause many cache misses. Unless those implementing the system are experts for the particular CPU's instruction set, the most reliable way to determine cache problems or floating-point unit stalls is to use performance tools. Intel provides a profiler, called VTune, that does give a lot of information, showing if cache misses or floating-point stalls have occurred. At a high level, a rearrangement of statements can help eliminate some of these problems; the necessity of rearranging is the result

of the optimizing compiler not being powerful enough to recognize the problems and rearrange transparently. But in many cases, a low-level solution is required, namely, writing parts of the code in assembly language. And once again portability becomes a problem.

All of these issues must be weighed and the trade-offs made when building a renderer. This is where the art of renderer construction really kicks in. Someone who does not understand all the issues will be unlikely to succeed in building a good renderer.

2.7.2 PORTABILITY VERSUS PERFORMANCE

This is a section that was not in the first edition of the book. I felt compelled to talk a little bit about this trade-off, especially since the Wild Magic engine has received some criticism for its choice of `Vector3` and `Matrix3` classes—they are not 16-byte friendly, which is a problem on consoles.

When a game company decides it wants to produce a new game, it invariably must decide on which platforms the game should run. Should it run universally on all platforms, including PCs, Macintosh, Playstation 2, Xbox, and Nintendo GameCube? What about on handheld devices? Moreover, if the next-generation consoles are to be released soon, should you develop to run on these (using your dev kits and beta hardware)? Supporting multiple platforms immediately drags you into the realm of portability versus performance.

As is true for any company, everything boils down to *time and resources*. Any business decision must take into account how much time it will take to build the product you envision and whether you have the resources to build it or you need to acquire more resources (funds, equipment, engineers, and so on).² When it comes to a particular platform, the developer-programmers who like mucking with all the technology invariably want to build everything themselves. Who better to squeeze every cycle out of a system for the ultimate in performance? When you support multiple platforms, though, is it cost-effective to write everything yourself? The executives, producers, and other nonprogrammer types typically prefer to minimize the costs, opting instead to purchase tools and license engine components. If these engine components run on multiple platforms, then the games may be written on top of the components to run on multiple platforms. However, such components may not be optimized for a particular platform, whether for speed or memory usage. The trade-off you must accept is that you give up some performance to be able to have your product run on multiple platforms.

On the other hand, if you choose to rely on licensing someone else's engine components, you must deal with the baggage that comes with them. They will have bugs, one or two that might be showstoppers for you. Hopefully, you can request

2. There is also *risk assessment*. When you decide to spend the time and resources developing a new product, you must assess the risk of not reaching your goal on time or not reaching your goal at all.

that they be fixed (in a timely manner), but regardless, this can affect the ship date of your products. You also must rely on the stability of the licensed component and the company producing that component. If you use a component, and the vendor decides to ship a major rewrite/update, that can affect your schedule. If the vendor decides to close its doors, that can really affect your schedule.

No one has ever said that running a business is a stress-free and easy thing to do!

Regarding the Wild Magic engine, I am in the same position of making trade-offs. The engine is of good quality, and I even use the description: It is a commercial-quality engine. However, *it is not a commercial engine*. The original goal for the engine was to provide source code implementations for various components you would find in a graphics engine, and it evolved to include some components you find in a physics engine. All this was designed for *educational purposes*. It is difficult to find working implementations of algorithms, especially ones that will not disappear after a week on the Web or ones whose author maintains regularly or doesn't abandon after a short period of time.

My business is education. My books are written to provide more depth on topics than you tend to find in the majority of computer graphics books. My clients are those people who purchase technical computer graphics books. Some work on Microsoft Windows PCs, some using Direct3D and some using OpenGL. Some work on Macintosh OS X machines using OpenGL. Some prefer a PC running a variation of Linux with hardware-accelerated drivers provided by their favorite graphics card manufacturer. Over the years, Wild Magic has evolved to support these platforms, and I have exchanged some performance to obtain portability. You might discover that some aspect of the engine could be faster if, instead of doing the computation on the CPU, it used Direct3D or OpenGL or Intel CPU extensions or AMD CPU extensions. Generally, this is not an oversight on my part, rather a consequence of my decision to be portable.

That said, Wild Magic 4 mainly involves a rewrite of the rendering system of the engine. I have attempted to abstract as much as possible to provide a platform-independent rendering API, yet one still having reasonable performance on all the supported platforms. The evolution of programmable graphics hardware and shader programming has made the abstraction and portability much simpler, while maintaining good performance. The scene graph management side of Wild Magic has also made the portability-versus-performance trade-off, but as you will see later in the book, the idea of *scene graph compilers* allows you to develop scenes in a platform-independent manner but compile them to platform-dependent data structures in an attempt to recapture performance.

2.8 API CONVENTIONS

Every graphics API has its conventions. And in many cases the APIs never agree on the same convention, which makes graphics programming quite ~~annoying~~ interesting. This section is devoted to comparing the conventions used by Wild Magic, OpenGL,

and Direct3D. Experimentation with the graphics APIs themselves gives you a lot of information about what conventions they use, but I have also used the OpenGL Red Book [SWND05] and the Direct3D documentation that ships with the freely downloadable DirectX SDK [Cor].

2.8.1 MATRIX REPRESENTATION AND STORAGE

Throughout this book I use mathematical notation to describe various concepts. This notation is *independent of graphics APIs*. What you have to do as a user of a graphics API is determine (1) how the mathematical expressions are implemented and (2) how quantities such as vectors and matrices are represented and stored in memory.

My mathematical conventions are as follows. I write 4-tuples (v_0, v_1, v_2, v_3) as 4×1 vectors and homogeneous matrices as shown:

$$\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}, \quad \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix}$$

To apply the matrix to the vector, I use

$$\begin{aligned} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} &= \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} \\ &= \begin{bmatrix} m_{00}v_0 + m_{01}v_1 + m_{02}v_2 + m_{03}v_3 \\ m_{10}v_0 + m_{11}v_1 + m_{12}v_2 + m_{13}v_3 \\ m_{20}v_0 + m_{21}v_1 + m_{22}v_2 + m_{23}v_3 \\ m_{30}v_0 + m_{31}v_1 + m_{32}v_2 + m_{33}v_3 \end{bmatrix} \end{aligned} \quad (2.114)$$

If you think of the matrix-vector product operation as a “black box” whose input is the 4-tuple (v_0, v_1, v_2, v_3) and whose output is the 4-tuple (p_0, p_1, p_2, p_3) , then all that matters is that Equation (2.114) is correctly implemented. Alternatively, the 4-tuples could be viewed as 1×4 vectors and homogeneous matrices as shown:

$$\begin{bmatrix} v_0 & v_1 & v_2 & v_3 \end{bmatrix}, \quad \begin{bmatrix} m_{00} & m_{10} & m_{20} & m_{30} \\ m_{01} & m_{11} & m_{21} & m_{31} \\ m_{02} & m_{12} & m_{22} & m_{32} \\ m_{03} & m_{13} & m_{23} & m_{33} \end{bmatrix}$$

To apply the matrix to the vector, you would use

$$[p_0 \ p_1 \ p_2 \ p_3] = [v_0 \ v_1 \ v_2 \ v_3] \begin{bmatrix} m_{00} & m_{10} & m_{20} & m_{30} \\ m_{01} & m_{11} & m_{21} & m_{31} \\ m_{02} & m_{12} & m_{22} & m_{32} \\ m_{03} & m_{13} & m_{23} & m_{33} \end{bmatrix} \quad (2.115)$$

where $p_i = \sum_{j=0}^3 m_{ij} v_j$ for $0 \leq i \leq 3$, just as in Equation (2.114). No matter which way you want to think of vectors—as column vectors or as row vectors—the product is the same 4-tuple. I will refer to the format of Equation (2.114) as having the *vector on the right*. The format of Equation (2.115) has the *vector on the left*.

The mathematical ways of formulating the application of a matrix to a vector say nothing about how the vectors or matrices are stored in computer memory. How these objects are stored is up to the graphics API and to you when you have such objects in your own source code. For example, consider 2-tuples and 2×2 matrices. The vector-on-the-right notation defines the matrix-vector product as

$$\begin{bmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \end{bmatrix} = \begin{bmatrix} m_{00}v_0 + m_{01}v_1 \\ m_{10}v_0 + m_{11}v_1 \end{bmatrix}$$

One implementation of the Vector2 class includes

```
class Vector2
{
private:
    float tuple[2];
public:
    float& operator[] (int i) { return tuple[i]; }
};
```

and one implementation of the Matrix2 class includes

```
class Matrix2
{
private:
    float m00, m01, m10, m11; // row-major order
public:
    Vector2 operator* (Vector2 V) const
    {
        Vector2 P;
        P[0] = m00*V[0] + m01*V[1];
        P[1] = m10*V[0] + m11*V[1];
        return P;
    }
};
```

Alternatively, an implementation of the `Matrix2` class could be

```
class Matrix2
{
private:
    float m00, m10, m01, m11; // column-major order
public:
    Vector2 operator* (Vector2 V) const
    {
        Vector2 P;
        P[0] = m00*V[0] + m01*V[1];
        P[1] = m10*V[0] + m11*V[1];
        return P;
    }
};
```

The order of storage of the abstract, doubly indexed elements of the 2×2 matrix is different between the two matrix classes, but the outputs of the matrix-vector product are the same. In the first example, the elements of the matrix are stored in what is called *row-major order*. In the second example, the elements are stored in what is called *column-major order*. Yet another possibility, if you want 16-byte alignment when working with SIMD CPUs or game consoles, is

```
class Vector2
{
private:
    float tuple[4]; // 16-bytes, tuple[2] = 0 and tuple[3] = 0 always
public:

    float& operator[] (int i) { return tuple[i]; }
    float Dot (Vector2 V) const
    {
        // The hardware loads tuple into a 16-byte register and loads
        // V.tuple into a 16-byte register and computes the dot product.
        return dot_product_of_tuple_and_V_tuple;
    }
};

class Matrix2
{
private:
    Vector2 row0, row1;
public:
    Vector2 operator* (Vector2 V) const
```

```

    {
        Vector2 P; // P[2] = 0 and P[3] = 0 are set by the constructor
        P[0] = row0.Dot(V);
        P[1] = row1.Dot(V);
        return P;
    }
};

```

In this implementation, the classes use additional memory that is always zeroed out, but takes advantage of hardware-assisted computations. The vectors are stored in linear memory as

```
v0 v1 0 0
```

and the matrices are stored in linear memory as

```
m00 m01 0 0 m10 m11 0 0
```

In all three implementations, the outputs of a matrix-vector operation all represent the same 2-tuple.

Generally, for 4-tuples and 4×4 matrices, the 16-byte alignment is achieved without padding. Essentially, we have four possibilities for storing the matrices and representing the vector-matrix product. The vector is either on the right or on the left, and the matrix is stored either in row-major order or in column-major order. The `Vector4` class and `Matrix4` classes are

```

class Vector4
{
private:
    float tuple[4];
public:
    float& operator[] (int i) { return tuple[i]; }
};

class Matrix4
{
private:
    float entry[16];
public:
    Vector4 operator* (Vector4 V) const
    {
        // What goes here? Possibilities are listed next.
    }
};

```

The possibilities are shown schematically with the matrix entries labeled according to their position within linear memory:

- Vector-on-right, row-major order (Wild Magic scene manager)

$$\begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

- Vector-on-right, column-major order (OpenGL)

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

- Vector-on-left, row-major order (Direct3D)

$$\begin{bmatrix} v_0 & v_1 & v_2 & v_3 \end{bmatrix} \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}$$

- Vector-on-left, column-major order

$$\begin{bmatrix} v_0 & v_1 & v_2 & v_3 \end{bmatrix} \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

Notice that the OpenGL and Direct3D products are the same when written as 4-tuples. The OpenGL Red Book [SWND05] shows that vectors are on the right and storage is in column-major order. However, you can also think of this as having the vector on the left and storing the matrix in row-major order. You may as well think of OpenGL and Direct3D using the same convention for vector-matrix multiplications and for matrix storage. To be consistent, the Wild Magic software renderer uses the same convention, even though the scene management system uses the vector-on-the-right convention. This consistency means that the vertex programs may be written to apply to all of my renderers. The matrices are passed as uniform constants in exactly the same manner for each of my renderers.

2.8.2 MATRIX COMPOSITION

Using the Wild Magic convention for matrix-vector products, if A and B are matrices to be applied to $\mathbf{V}$, A first and B second, then the composition is $\mathbf{P} = B(A\mathbf{V}) = (BA)\mathbf{V}$. You start with the rightmost quantity and apply operations from right to left. Chapter 4 discusses scene graph management. A fundamental part of this is a hierarchy of nodes, each node having a transformation M_p that maps it to the world coordinate system and each child node having a transformation M_c that maps it to its parent's coordinate system. The transformation to map the child to the world is $M_p M_c$. When applied to a vector $\mathbf{V}$ in the child's model space, the corresponding vector in world space is $M_p M_c \mathbf{V}$. The rule for composition is applied here. The actual matrix-matrix products occur in the scene graph geometric update function.

OpenGL uses the same convention for composition that Wild Magic uses. The product $P = BA$ is computed using

```
float A[16] = <some 4x4 matrix>;
float B[16] = <some 4x4 matrix>;
float P[16];
glMatrixMode(GL_MODELVIEW); // Use the model-view matrix stack.
glPushMatrix(); // Save the current matrix.
glLoadMatrixf(B);
glMultMatrixf(A);
glGetFloatv(GL_MODELVIEW_MATRIX,P);
glPopMatrix(); // Restore the current matrix to its original value.
```

Direct3D uses the convention $\mathbf{P} = (\mathbf{V}A)B = \mathbf{V}(AB)$, where it is understood that $\mathbf{V}$ is a 1×3 row vector. The product $P = AB$ is computed using

```
D3DXMATRIX A = <some 4x4 matrix>;
D3DXMATRIX B = <some 4x4 matrix>;
D3DXMATRIX P;
D3DXMatrixMultiply(&P,&A,&B);
```

Since the scene graph management system and abstract renderer layer hide these details, there is no reason to be concerned about the composition. All front-end matrix and vector calculations use vector-on-the-right and row-major storage, and the composition AB refers to applying the product as $AB\mathbf{V} = A(B\mathbf{V})$.

2.8.3 VIEW MATRICES

Let us now compare the view matrices of the graphics APIs. The camera eye position is $\mathbf{E}$; the camera view direction is $\mathbf{D}$; the camera up direction is $\mathbf{U}$; and the camera right direction is $\mathbf{R}$ —all thought of as 3×1 column vectors. The transposes are $\mathbf{D}^T$,

$\mathbf{U}^T$, and $\mathbf{R}^T$ —all thought of as 1×3 row vectors. The vector $\mathbf{0}$ denotes the 3×1 zero vector and $\mathbf{0}^T$ is the 1×3 zero vector.

The Direct3D view matrix is

$$H_{\text{view}} = \left[\begin{array}{ccc|c} \mathbf{R} & \mathbf{U} & \mathbf{D} & \mathbf{0} \\ \hline -\mathbf{R}^T \mathbf{E} & -\mathbf{U}^T \mathbf{E} & -\mathbf{D}^T \mathbf{E} & 1 \end{array} \right]$$

Recall that to apply the matrix to a vector, the vector is on the left and written as a row vector. The matrix itself is stored in row-major order. I chose $\{\mathbf{D}, \mathbf{U}, \mathbf{R}\}$ to be a right-handed orthonormal set, so $\{\mathbf{R}, \mathbf{U}, \mathbf{D}\}$ is a left-handed orthonormal set. As the Direct3D documentation [Cor] states, the camera coordinate system is left-handed. I have chosen the Wild Magic view matrix to be the same matrix.

The OpenGL Red Book [SWND05] states that the default camera settings include the eye position at the origin, the up vector in the positive y -direction $(0, 1, 0)$, the right vector in the positive x -direction $(1, 0, 0)$, and the view vector in the negative z -direction $(0, 0, -1)$. Probably the most common way to set the view matrix to different values is by calling `gluLookAt`. A call to this function generates the OpenGL view matrix, shown in row-major order for the vector-on-the-left convention:

$$H_{\text{view,ogl}} = \left[\begin{array}{ccc|c} \mathbf{R} & \mathbf{U} & -\mathbf{D} & \mathbf{0} \\ \hline -\mathbf{R}^T \mathbf{E} & -\mathbf{U}^T \mathbf{E} & \mathbf{D}^T \mathbf{E} & 1 \end{array} \right]$$

The set $\{\mathbf{R}, \mathbf{U}, -\mathbf{D}\}$ is right-handed, but the last vector is the negated direction of view. I do not know why this was chosen, but perhaps it was to ensure that $Q = [\mathbf{R} \ \mathbf{U} \ -\mathbf{D}]$ is a rotation matrix. By the way, the Q for Direct3D and Wild Magic is a *reflection matrix*, but as a transformation applied to a left-handed coordinate system, it is a *rotation transformation*. If you have had training in linear algebra, you should recall that a matrix and a transformation are not the same thing. Given a transformation, you can represent it by a matrix, but that matrix depends on the bases chosen for the domain and range of the transformation.

In my implementations, I avoid the use of Direct3D's utility functions `D3DXDMATRIXLookAt*` and OpenGL's utility function `gluLookAt`. I use the view matrix H_{view} for all of the renderers. The base class `Renderer` uses the following code to set the view matrix:

```
const Vector3f& rkEye = m_pkCamera->GetLocation();
const Vector3f& rkRVector = m_pkCamera->GetRVector();
const Vector3f& rkUVector = m_pkCamera->GetUVector();
const Vector3f& rkDVector = m_pkCamera->GetDVector();

m_kViewMatrix[0][0] = rkRVector[0];
m_kViewMatrix[0][1] = rkUVector[0];
m_kViewMatrix[0][2] = rkDVector[0];
m_kViewMatrix[0][3] = 0.0f;
```

```

m_kViewMatrix[1][0] = rkRVector[1];
m_kViewMatrix[1][1] = rkUVector[1];
m_kViewMatrix[1][2] = rkDVector[1];
m_kViewMatrix[1][3] = 0.0f;
m_kViewMatrix[2][0] = rkRVector[2];
m_kViewMatrix[2][1] = rkUVector[2];
m_kViewMatrix[2][2] = rkDVector[2];
m_kViewMatrix[2][3] = 0.0f;
m_kViewMatrix[3][0] = -rkRVector.Dot(rkEye);
m_kViewMatrix[3][1] = -rkUVector.Dot(rkEye);
m_kViewMatrix[3][2] = -rkDVector.Dot(rkEye);
m_kViewMatrix[3][3] = 1.0f;

```

The derived class `SoftRenderer` has no additional code to go with this. The derived class `Dx9Renderer` follows up with

```
m_pqDevice->SetTransform(D3DTS_VIEW, (D3DXMATRIX*)(float*)m_kViewMatrix);
```

where `m_pqDevice` is of type `IDirect3DDevice9*`. The derived class `OpenGLRenderer` follows up with

```

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf((const float*)m_kViewMatrix);

```

2.8.4 PROJECTION MATRICES

The Direct3D perspective projection matrix follows, written in the format for the vector-on-the-left convention:

$$H_{\text{proj}} = \left[\begin{array}{ccc|c} \frac{2d_{\min}}{r_{\max}-r_{\min}} & 0 & 0 & 0 \\ 0 & \frac{2d_{\min}}{u_{\max}-u_{\min}} & 0 & 0 \\ \frac{-(r_{\max}+r_{\min})}{r_{\max}-r_{\min}} & \frac{-(u_{\max}+u_{\min})}{u_{\max}-u_{\min}} & \frac{d_{\max}}{d_{\max}-d_{\min}} & 1 \\ \hline 0 & 0 & \frac{-d_{\max}d_{\min}}{d_{\max}-d_{\min}} & 0 \end{array} \right]$$

I have chosen the Wild Magic view matrix to be the same matrix.

The OpenGL projection matrix is different, written here in row-major format for the vector-on-the-left convention:

$$H_{\text{proj,ogl}} = \left[\begin{array}{ccc|c} \frac{2d_{\min}}{r_{\max}-r_{\min}} & 0 & 0 & 0 \\ 0 & \frac{2d_{\min}}{u_{\max}-u_{\min}} & 0 & 0 \\ \frac{r_{\max}+r_{\min}}{r_{\max}-r_{\min}} & \frac{u_{\max}+u_{\min}}{u_{\max}-u_{\min}} & \frac{-(d_{\max}+d_{\min})}{d_{\max}-d_{\min}} & -1 \\ \hline 0 & 0 & \frac{-2d_{\max}d_{\min}}{d_{\max}-d_{\min}} & 0 \end{array} \right]$$

Because of the convention used in the OpenGL view matrix, namely, $Q = [\mathbf{R} \ \mathbf{U} \ -\mathbf{D}]$, the view coordinates are $\mathbf{X}_{\text{view}} = (r, u, -d)$. The application of the projection matrix effectively undoes the sign change on the d -component (and on the $\mathbf{D}$ -vector). OpenGL maps the d -components to the interval $[-1, 1]$, explaining the slightly different terms in the third columns of the matrices H_{proj} and $H_{\text{proj,ogl}}$.

Once again in my implementations, I avoid the use of Direct3D's utility functions `D3DXMatrixPerspective*` and OpenGL's utility function `glFrustum`. I use the view matrix H_{proj} for all of the renderers. The base class `Renderer` uses the following code to set the projection matrix:

```
float fRMin, fRMax, fUMin, fUMax, fDMin, fDMax;
m_pkCamera->GetFrustum(fRMin,fRMax,fUMin,fUMax,fDMin,fDMax);

float fInvRDiff = 1.0f/(fRMax - fRMin);
float fInvUDiff = 1.0f/(fUMax - fUMin);
float fInvDDiff = 1.0f/(fDMax - fDMin);

if (m_pkCamera->Perspective)
{
    m_kProjectionMatrix[0][0] = 2.0f*fDMin*fInvRDiff;
    m_kProjectionMatrix[0][1] = 0.0f;
    m_kProjectionMatrix[0][2] = 0.0f;
    m_kProjectionMatrix[0][3] = 0.0f;
    m_kProjectionMatrix[1][0] = 0.0f;
    m_kProjectionMatrix[1][1] = 2.0f*fDMin*fInvUDiff;
    m_kProjectionMatrix[1][2] = 0.0f;
    m_kProjectionMatrix[1][3] = 0.0f;
    m_kProjectionMatrix[2][0] = -(fRMin + fRMax)*fInvRDiff;
    m_kProjectionMatrix[2][1] = -(fUMin + fUMax)*fInvUDiff;
    m_kProjectionMatrix[2][2] = fDMax*fInvDDiff;
    m_kProjectionMatrix[2][3] = 1.0f;
    m_kProjectionMatrix[3][0] = 0.0f;
    m_kProjectionMatrix[3][1] = 0.0f;
    m_kProjectionMatrix[3][2] = -fDMax*fDMin*fInvDDiff;
    m_kProjectionMatrix[3][3] = 0.0f;
}
```



```

else
{
    m_kProjectionMatrix[0][0] = 2.0f*fInvRDiff;
    m_kProjectionMatrix[0][1] = 0.0f;
    m_kProjectionMatrix[0][2] = 0.0f;
    m_kProjectionMatrix[0][3] = 0.0f;
    m_kProjectionMatrix[1][0] = 0.0f;
    m_kProjectionMatrix[1][1] = 2.0f*fInvUDiff;
    m_kProjectionMatrix[1][2] = 0.0f;
    m_kProjectionMatrix[1][3] = 0.0f;
    m_kProjectionMatrix[2][0] = 0.0f;
    m_kProjectionMatrix[2][1] = 0.0f;
    m_kProjectionMatrix[2][2] = fInvDDiff;
    m_kProjectionMatrix[2][3] = 0.0f;
    m_kProjectionMatrix[3][0] = -(fRMin + fRMax)*fInvRDiff;
    m_kProjectionMatrix[3][1] = -(fUMin + fUMax)*fInvUDiff;
    m_kProjectionMatrix[3][2] = -fDMin*fInvDDiff;
    m_kProjectionMatrix[3][3] = 1.0f;
}

```

The derived class `SoftRenderer` has no additional code to go with this. The derived class `Dx9Renderer` follows up with

```

m_pqDevice->SetTransform(D3DTS_PROJECTION,
                        (D3DXMATRIX*)(float*)m_kProjectionMatrix);

```

where `m_pqDevice` is of type `IDirect3DDevice9*`. The derived class `OpenGLRenderer` follows up with

```

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glMultMatrixf((const float*)m_kProjectionMatrix);

```

The base class `Renderer` has a conditional statement, setting the projection matrix to be perspective in one case but orthographic in the other. The orthographic matrix is chosen to be the same for all the graphics APIs:

$$H_{\text{orthproj}} = \left[\begin{array}{ccc|c} \frac{2}{r_{\max}-r_{\min}} & 0 & 0 & 0 \\ 0 & \frac{2}{u_{\max}-u_{\min}} & 0 & 0 \\ 0 & 0 & \frac{1}{d_{\max}-d_{\min}} & 0 \\ \hline -\frac{r_{\max}+r_{\min}}{r_{\max}-r_{\min}} & -\frac{u_{\max}+u_{\min}}{u_{\max}-u_{\min}} & -\frac{d_{\min}}{d_{\max}-d_{\min}} & 1 \end{array} \right]$$

One use for an orthographic matrix is to allow you to draw *screen-space polygons*. These are 2D geometric objects that are drawn either as background polygons or as foreground polygons. The latter case is common when you want a full-screen game application that needs a user interface. For example, you can create menus, control bars, sliders, edit controls, and so on, to be drawn on top of the 3D scene.

2.8.5 WINDOW HANDEDNESS

OpenGL thinks of the window coordinates as being right-handed. The origin (0, 0) is the lower-left corner of the window. The x -values increase as you move from left to right. The y -values increase as you move from bottom to top. The mapping from NDC values (r'' , u'') to (x , y) is shown in Equation (2.71); namely,

$$\begin{aligned} x &= \left(\frac{1-r''}{2}\right) p_\ell W + \left(\frac{1+r''}{2}\right) p_r W = \frac{W}{2} [(p_r + p_\ell) + (p_r - p_\ell)r''] \\ y &= H - \left[\left(\frac{1-u''}{2}\right) p_b H + \left(\frac{1+u''}{2}\right) p_t H\right] \\ &= \frac{H}{2} [(2 - p_t - p_b) + (p_b - p_t)u''] \end{aligned}$$

This transformation is computed internally in OpenGL. You set the viewport parameters using the OpenGL function `glViewport`, as shown in the function `OpenGLRenderer::OnViewportChange`:

```
GLint iX = (GLint)(fPortL*m_iWidth);
GLint iY = (GLint)(fPortB*m_iHeight);
GLsizei iW = (GLsizei)((fPortR - fPortL)*m_iWidth);
GLsizei iH = (GLsizei)((fPortT - fPortB)*m_iHeight);
glViewport(iX,iY,iW,iH);
```

On the other hand, Direct3D thinks of the window coordinates as being left-handed. The origin (0, 0) is the upper-left corner of the window. The x -values increase as you move from left to right. The y -values increase as you move from top to bottom. The transformation from NDC to window coordinates is handled by the renderer internally. You set the viewport parameters using the Direct3D function `SetViewport`, as shown in the function `Dx9Renderer::OnViewportChange`:

```
D3DVIEWPORT9 kViewport;
kViewport.X = (DWORD)(fPortL*m_iWidth);
kViewport.Y = (DWORD)((1.0f - fPortT)*m_iHeight);
kViewport.Width = (DWORD)((fPortR - fPortL)*m_iWidth);
kViewport.Height = (DWORD)((fPortT - fPortB)*m_iHeight);
```

```
kViewport.MinZ = 0.0f;
kViewport.MaxZ = 1.0f;
m_pqDevice->SetViewport(&kViewport);
```

The x - and width values are handled the same as in OpenGL, but the y - and height values are handled differently. Well, the height value looks the same, but mathematically it is computed as shown next, leading to the same equation.

Figure 2.50 compares the representation of windows and viewports for OpenGL and Direct3D. The port values p_t and p_b must be reflected to $(1 - p_t)$ and $(1 - p_b)$ to make the Direct3D viewport right-handed. The NDC-to-viewport mapping for the y -coordinate is therefore

$$\begin{aligned} y &= \left(\frac{1 - u''}{2} \right) ((1 - p_b)H) + \left(\frac{1 + u''}{2} \right) ((1 - p_t)H) \\ &= \frac{H}{2} [(2 - p_t - p_b) + (p_b - p_t)u''] \end{aligned}$$

Notice that $u'' = 1$ maps to $y = (1 - p_t)H$ and $u'' = -1$ maps to $y = (1 - p_b)H$. Indeed, this is the same mapping as for OpenGL, which is what you should expect. An NDC point (r'', u'') means the same thing, whether in OpenGL or in Direct3D; after all, it is a “normalized device coordinate” intended to be independent of graphics API or physical device. This point should map to the same pixel in either graphics API. Finally, getting back to my comment about the viewport height h , in this setting it is computed by

$$h = (1 - p_b)H - (1 - p_t)H = (p_t - p_b)H$$

where H is the window height. The actual source code computes the rightmost expression, but the middle expression is really what Figure 2.50 indicates.

The Wild Magic software renderer also has a function `SoftRenderer::OnViewportChange`, which implements

```
float fHalfWidth = 0.5f*m_iWidth;
float fHalfHeight = 0.5f*m_iHeight;
m_fXCoeff0 = fHalfWidth*(fPortR + fPortL);
m_fXCoeff1 = fHalfWidth*(fPortR - fPortL);
m_fYCcoeff0 = fHalfHeight*(2.0f - fPortT - fPortB);
m_fYCcoeff1 = fHalfHeight*(fPortB - fPortT);
```

This is a direct implementation of Equation (2.71).

2.8.6 ROTATIONS

As discussed in Section 2.8.1, my convention for matrix-vector multiplication is to store the matrix in row-major order and to place the vector on the right (a column

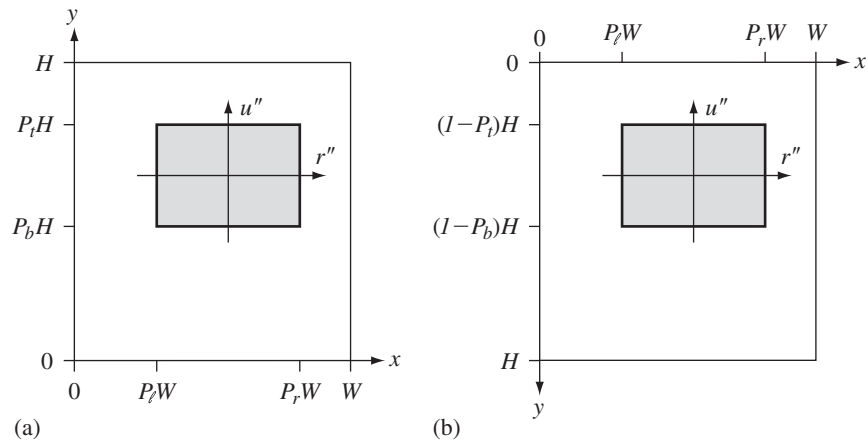


Figure 2.50 (a) OpenGL uses right-handed window coordinates. (b) Direct3D uses left-handed window coordinates.

vector) for matrix times the vector. If R is a 3×3 rotation matrix and $\mathbf{V}$ is a 3×1 vector, the rotated vector is $R\mathbf{V}$. However, there is an additional convention that must be chosen regarding which direction you rotate about the axis of rotation when the input angle is a positive number.

The convention in Wild Magic was discussed in Section 2.2.1, in the subsection entitled “Rotation.” The direction of rotation is counterclockwise about the rotation axis when viewed by an observer who is on the positive side of the plane perpendicular to the axis, meaning on the side to which the rotation axis direction points and looking at the plane with view direction in the negative of the rotation axis direction.

OpenGL allows you to rotate via the functions

```
void glRotatef (float angle, float x, float y, float z);
void glRotated (double angle, double x, double y, double z);
```

If you want to see the rotation matrix itself, try

```
Vector3f kAxis(1.0f,1.0f,1.0f);
kAxis.Normalize();
float fRadians = 0.1f;
float fDegrees = Mathf::RAD_TO_DEG*fRadians;
Matrix3f kWildMagicRotate(kAxis,fRadians);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

```
glRotatef(fDegrees,kAxis.X(),kAxis.Y(),kAxis.Z());
float afOpenGLRotate[16];
glGetFloatv(GL_MODELVIEW_MATRIX,afOpenGLRotate);
```

The resulting rotation matrix is, of course, 4×4 , but the upper-left 3×3 matrix is the same one used by Wild Magic, as shown in Equation (2.22). For the sample code here, the rotation matrix is

$$\left[\begin{array}{ccc|c} 0.9966 & -0.0559 & 0.0593 & 0 \\ 0.0593 & 0.9966 & -0.0559 & 0 \\ -0.0559 & 0.0593 & 0.9966 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array} \right]$$

This is in the form in which you would multiply a vector by placing it on the right of the matrix as a column vector.

A couple of items to be aware of. First, OpenGL's `glRotate` functions expect the angle in degrees. Wild Magic wants radians. Second, Wild Magic expects the rotation axis vector to be unit length. You can pass a non-unit-length vector to `glRotate` and it will compute the correct rotation matrix. This means OpenGL internally normalizes the axis vector.

Direct3D allows you to rotate via the function

```
D3DXMATRIX* D3DXMatrixRotationAxis (
    D3DXMATRIX* pOut, const D3DVECTOR3* pV, FLOAT angle);
```

The input angle must be in radians. The DirectX documentation [Cor] states “Angles are measured clockwise when looking along the rotation axis toward the origin.” At first glance, you might think that the rotation uses the convention opposite that of Wild Magic and OpenGL. In fact, it uses the same convention, because Direct3D uses the vector-on-the-left convention. The following code

```
Vector3f kAxis(1.0f,1.0f,1.0f);
kAxis.Normalize();
float fRadians = 0.1f;
Matrix3f kWildMagicRotate(kAxis,fRadians);

D3DXMATRIX kRotation;
D3DXMatrixRotationAxis(&kRotation,(D3DVECTOR3*)(float*)kAxis,fRadians);
```

produces the matrix

$$\left[\begin{array}{ccc|c} 0.9666 & 0.0593 & -0.0559 & 0 \\ -0.0559 & 0.9666 & 0.0593 & 0 \\ 0.0593 & -0.0559 & 0.9966 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array} \right]$$

This is the transpose of what was produced by OpenGL and Wild Magic, but these APIs use the vector-on-the-right convention. Since Direct3D uses the vector-on-the-left convention, in fact, the rotation applied to a vector produces the same vector in all three APIs.

2.8.7 FAST COMPUTATIONS USING THE GRAPHICS API

One of the criticisms of Wild Magic has been that it does not take advantage of CPU-specific instructions for fast vector and matrix arithmetic. For example, it neither ships with support for Intel's Streaming SIMD Extensions (SSE) Driver, nor with support for AMD's 3D Now! extensions. Neither does it ship with Multi-Media Extensions (MMX). In fact, it does not ship with any SIMD AltiVec extensions for the PowerPC Macintosh computers. This is not an oversight on my part. The main goal for my engine is to illustrate ideas you encounter in graphics and game programming. I have called the engine a "commercial-quality engine," a descriptor I mean to be different from that of "commercial engine." Portability is my major criterion for maintaining the engine because I am interested in reaching the largest number of readers possible, choosing not to be solely a PC engine running on Microsoft Windows with DirectX. This alone takes a lot of time; adding in a few more hardware-specific components will take more maintenance time than I care to give. Should I decide to have support for every possible piece of hardware people care about, I will make the engine commercial, which means not providing it on a CD-ROM of a book.

Despite this, you can take advantage of any hardware support for vector and matrix computations if the graphics drivers do so.

OpenGL appears not to have in its interface the ability to explicitly multiply a matrix times a vector, giving you access to the result. Naturally, matrices are applied to the vectors in a vertex array for the purpose of rendering. Should you find the need to multiply vectors, you can pack four vectors at a time into a matrix, use the matrix stack to multiply, and then read back the results.

```
glMatrixMode(GL_MODELVIEW);
<apply your transformations>;

int iVQuantity = <number of vectors, assume it is a multiple of 4>;
Vector4f* akInput = <array of vectors>;
Vector4f* akOutput = <array of transformed vectors>;
```

```

int iPackedQuantity = iVQuantity/4;
float* afPackedInput = (float*)akInput;
float* afPackedOutput = (float*)akOutput;
for (int i = 0; i < iPackedQuantity; i++)
{
    glPushMatrix();
    glMultMatrix3f(afPackedInput);
    glGetFloatv(GL_MODELVIEW_MATRIX,afPackedOutput);
    glPopMatrix();
}

```

EXERCISE
2.17

Modify the code sample to support an array of vectors for which the quantity is not a multiple of 4. (Alternatively, you could arrange for your vector arrays to be padded to a multiple of 4.) For the case when you do have a multiple of 4 vectors, write a test program and calculate the time it takes to use the built-in Wild Magic matrix-vector operations for transforming. Also calculate the time it takes OpenGL to do the computations as shown here. If there is a significant speedup, a real application using OpenGL would want the Transformation class of Wild Magic to be reimplemented to use OpenGL. ■

Direct3D does have explicit functions to support vector and matrix operations. The transformation of a vector by a matrix is

```

D3DXVECTOR4* D3DXVec4Transform (
    D3DXVECTOR4* pOut, CONST D3DXVECTOR4* pV, CONST D3DXMATRIX* pM);

```

and the transformation of an array of vectors by a matrix is

```

D3DXVECTOR4* D3DXVec4TransformArray (
    D3DXVECTOR4* pOut, UINT OutStride, CONST D3DXVECTOR4* pV,
    UINT VStride, CONST D3DXMATRIX* pM, UINT n);

```

In the previous section, we had an example of a rotation about $(1, 1, 1)/\sqrt{3}$ by an angle of 0.1 radians. If you add more code to that sample,

```

Vector3f kAxis(1.0f,1.0f,1.0f);
kAxis.Normalize();
float fRadians = 0.1f;
float fDegrees = Mathf::RAD_TO_DEG*fRadians;
Matrix3f kRot(kAxis,fRadians);

D3DXMATRIX kRotation;
D3DXMatrixRotationAxis(&kRotation,(D3DXVECTOR3*)(float*)kAxis,fRadians);

```

```
D3DXVECTOR4 kInput, kOutput;
kInput.x = 1.0f; kInput.y = 0.0f; kInput.z = 0.0f; kInput.w = 0.0f;
D3DXVec4Transform(&kOutput,&kInput,&kRotation);
```

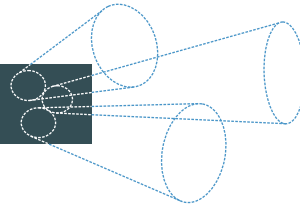
the output vector is (0.9966, 0.0593, -0.0559, 0), which is the first row of the Direct3D rotation matrix as expected. A sample usage for the transformation of an array is

```
D3DXVECTOR4 akInput[4], akOutput[4];
akInput[0].x = 1.0f; akInput[0].y = 0.0f;
    akInput[0].z = 0.0f; akInput[0].w = 0.0f;
akInput[1].x = 0.0f; akInput[1].y = 1.0f;
    akInput[1].z = 0.0f; akInput[1].w = 0.0f;
akInput[2].x = 0.0f; akInput[2].y = 0.0f;
    akInput[2].z = 1.0f; akInput[2].w = 0.0f;
akInput[3].x = 0.0f; akInput[3].y = 0.0f;
    akInput[3].z = 0.0f; akInput[3].w = 1.0f;
D3DXVec4TransformArray(akOutput,sizeof(D3DXVECTOR4),akInput,
    sizeof(D3DXVECTOR4),&kRotation,4);
```

The output array has vectors equal to the four rows of the rotation matrix.

EXERCISE 2.18

Write a test program and calculate the time it takes to use the built-in Wild Magic matrix-vector operations for transforming. Also calculate the time it takes Direct3D to do the computations as shown here. If there is a significant speedup, a real application using Direct3D would want the Transformation class of Wild Magic to be reimplemented to use Direct3D. ■



RENDERERS

Now that we have seen the basics for a graphics system, let us go through the process of drawing geometric primitives in greater detail. Many of these details are handled for you by the drivers that ship with hardware-accelerated graphics cards. Some feedback was solicited from reviewers of the first edition of this book in order to identify what improvements could be made. One of the comments was to focus less on software rendering because graphics hardware is so abundant and most developers will not be exposed to that type of detail when using a graphics API. I have taken the opposite stance for three reasons:

1. Someone has to write the graphics drivers; that might be you. This is not to say that the decisions you make when designing and building drivers for hardware are the same as those for software. On fast dedicated hardware, you tend to choose algorithms that are “dumb and fast”; whereas, on CPUs you typically select algorithms that are “smart” about use of the CPU resources. However, the graphics fundamentals are the same in either case—it is just a question of what your target hardware is (CPU or GPU).
2. Cell phone and mobile technology is ever evolving. New embedded devices might require low-level graphics development, at least until those particular devices are enhanced with hardware support for the graphics.
3. Working through a software renderer—and perhaps more important, writing one—gives you a greater appreciation for what the graphics APIs are really doing for you. As simple as it might seem, when you write a software renderer, you will discover that there are a lot of decisions to make and a lot of details to tend to.

If anything, hopefully you will get an appreciation for the level of effort a graphics hardware company must put into writing graphics drivers. We are all guilty of cursing about the bugs we discover in those drivers. What comes to my mind, though, is something about “walk a mile in my shoes”

The CD-ROM contains a fully functioning software renderer that is based on shader programs. All the sample projects allow you to compile for an OpenGL renderer, a Direct3D renderer, and a software renderer. By no means is this software renderer optimized for speed. My intent is to illustrate the ideas without complicating the code with cryptic routines whose sole intent is to squeeze out a few more cycles. That said, feel free to start replacing a piece of the renderer at a time with optimized modules. For example, I do not use a Bresenham’s line-drawing algorithm for rasterizing the wireframe of a triangle mesh or for rasterizing polylines. The code does determine the correct direction in which to iterate (x - or y -direction), but it uses floating-point arithmetic to evaluate the equation of the line segment connecting points. There is no attempt to avoid the expensive conversion from a floating-point number to an integer. This alone can lead to a significant speedup in a software renderer. Mipmapping is computed on a per-pixel basis. Sometimes you will find that software renderers use cheaper alternatives, such as per-scan-line or per-triangle mipmapping, but these come at a loss of quality in the rendered scene.

The shader programs to illustrate the concepts were written using Cg from NVIDIA. Most were compiled with Cg 1.4, but some used Cg 1.1 to circumvent some bugs in the 1.4 version. The compiler output is text containing assembly instructions, both for OpenGL and Direct3D. Wild Magic treats these as one of the renderer resources and loads these from disk as needed. I made this choice rather than relying on the Cg Runtime environment so that you are not forced to use a system when you prefer something else. Had I written Wild Magic for a commercial environment, and that environment used Cg or its equivalent, I would most definitely use whatever came with the environment. A consequence of my choice is that you just as readily write HLSL programs for Direct3D and compile them using the command-line HLSL compiler. Naturally, my choice means that I do not use the shader run-time environments provided by Direct3D or OpenGL 2.0. The Direct3D output files have a compound extension, `dx9.wmsp`; the OpenGL output files have a compound extension, `ogl.wmsp`; and the software renderer output files have a compound extension, `sft.wmsp`. I wrote the software shader programs myself, manually translating the Cg programs to C++ code. Nothing prevents you, though, from writing a translator to automatically convert to C++ or even to invent your own software graphics assembly language and write a compiler that converts Cg programs to your assembly.

Because my intent is to illustrate and educate rather than provide you with all the bells and whistles that a graphics system can have, the rendering libraries have the ability neither to procedurally generate shader programs nor to stitch existing shaders together. These are features you will find in commercial engines. The amount of work to add these to Wild Magic is not that daunting, but will require you to choose someone’s run-time environment in order to compile the generated/stitched shader programs. Alternatively, you can shell out to the operating system and call the Cg compiler or HLSL compiler to produce the text files containing the assembly

instructions, and then rely on Wild Magic to load them from disk to memory. My development plan is to incorporate support for generating and stitching shaders in a later minor version of Wild Magic 4.

3.1 SOFTWARE RENDERING

The job of the scene graph management system is to identify the potentially visible set of objects in each scene graph when drawing is to be performed. This process is described in Section 4.5. Once created, an iteration is made over all the objects in this set. The geometric primitive associated with each object is sent to the renderer for drawing. Let's start by considering a geometric primitive that is a single triangle. The assumption is made that all the resources needed by the drawing operation have been given to the renderer. The details of how this is done is the topic of Section 3.3.8. All function references in this section are to the software renderer source code on the CD-ROM. The class name is `SoftRenderer`, but I will omit this in the discussion unless I need to make it clear whether `SoftRenderer` must make a computation or its base class `Renderer` does a computation.

The entry point into the drawing system of the software renderer is the function `DrawElements`. You can follow along if you like by writing a simple application to draw a triangle and step through with the debugger! The entry point is quite small:

```
void DrawElements ()
{
    ApplyVertexShader();
    (this->*m_aoDrawFunction[m_pkGeometry->Type])();
}
```

3.1.1 VERTEX SHADERS

The triangle has three vertices, each with a position and optional vertex attributes. Some of the vertex attributes need to be computed first; for example, if you are using dynamic lighting, the vertices have been assigned vertex normals, but the vertex colors must be computed. In fact, these colors might be blended with colors from a texture image, in which case the vertices also have texture coordinates assigned to them. The first step in the drawing process is to apply a *vertex shader* to each vertex. This amounts to calling a *vertex program* whose inputs are the vertex position and other attributes. The positions are typically passed as model-space coordinates, but this is not necessary. Also passed to the vertex program are matrices to be used for transforming the vertex positions, normals, and other quantities. The program minimally computes the clip-space coordinates of the vertex positions and returns these as outputs to be used by the clipper and rasterizer. If the model-space positions are passed to the program, then you will also pass the matrix for transforming from model space to clip space. The program can return additional outputs, some of

them just as a *pass-through* (texture coordinates or vertex colors passed as inputs are returned as outputs), but others are generated by the program (tangent, normal, and bitangent for bump-mapping; texture coordinates for projected texturing).

As an example, a Cg program for vertex coloring is listed next. You can find this on the CD-ROM together with other Cg shader programs in the directory

GeometricTools\WildMagic4\Data\ShaderPrograms\Cg

The file of interest is VertexColor3.cg. A vertex program in that file is

```
void VertexColor3VProgram
(
    in float4      kModelPosition : POSITION,
    in float3      kModelColor : COLOR,
    out float4      kClipPosition : POSITION,
    out float3      kVertexColor : COLOR,
    uniform float4x4 WVPMatrix)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    // Pass through the vertex color.
    kVertexColor = kModelColor;
}
```

The program has two inputs. The first input is the vertex position in model-space coordinates. The second input is the vertex color (red, green, blue, but no alpha). The uniform matrix `WVPMatrix` is the homogeneous transformation that maps model-space coordinates to clip-space coordinates.

The first line of code transforms the input position. The matrix itself is the concatenation of a few matrices found in the geometric pipeline; see Equation (2.74), but according to the discussion about API conventions of Section 2.8, my OpenGL, Direct3D, and software renderers all pass matrices to the vertex programs so that the vector-on-the-left convention is used. Thus,

$$H_{\text{wvp}} = H_{\text{world}} H_{\text{view}} H_{\text{proj}}$$

is the matrix representing `WVPMatrix`, where the three matrices on the right are stored for the vector-on-the-left convention (the mathematically written matrices of Section 2.3.7 are all transposed). The transformation is

$$\mathbf{X}_{\text{clip}} = \mathbf{X}_{\text{model}} H_{\text{wvp}}$$

where $\mathbf{X}_{\text{model}}$ is the 1×4 vector that represents `kModelPosition`. The w -value for the input point is 1. The (x, y, z) -values for the input point are the model-space coordinates for the vertex position. The point $\mathbf{X}_{\text{clip}}$ is the 1×4 vector that represents

`kClipPosition`. If you were to write the HLSL equivalent of this vertex program, your line of code for transforming the input point would be the same, placing the vector on the left and the matrix on the right in the `mul` operation.

The second line of code in the vertex program is a pass-through of the vertex color. The rasterizer will interpolate both `kClipPosition` and `kVertexColor` to assign colors to the pixels making up the triangle.

The vertex program call occurs in a shallow wrapper called `ApplyVertexShader`. At the time of call, the renderer already has an array of vertices to process, and it knows the size of each vertex. The vertex program outputs a chunk of data for each input vertex. The size of that chunk is the same for all the vertices, but what it contains is based on the vertex program itself. For example, if your input is a vertex position, a vertex normal, and information for dynamic lighting (material colors, light parameters, camera parameters, and so on), the output will be the clip-space position and a vertex color. The input has a 3-tuple position and a 3-tuple normal; the output has a 4-tuple position and a 4-tuple color (includes an alpha channel). The software renderer supplies the storage for the outputs, so `ApplyVertexShader` increases the size of the current output array (if necessary) to store the results passed back from the vertex program. The actual line of code that calls the vertex program is

```
m_oVProgram(afRegister,afInVertex,afOutVertex);
```

The first parameter is an array of constants. In our previous example, this array stores the matrix H_{wvp} . The second parameter is the input array of vertices (model-space positions and attributes). The third parameter is the output array of vertices (clip-space positions and attributes).

Once the output storage is guaranteed to be large enough, the input vertices are iterated, each one passed to the vertex program, and the outputs stored in the output array. The software renderer remembers how many outputs are stored. Eventually, the clipper receives this output array and potentially generates more vertices if the triangle must be truncated by one or more frustum planes. The clipper must know how many vertices are currently in use and then increase the output array size as needed during clipping. New vertices generated by clipping are appended to the current list of output vertices.

The second function call in `DrawElements` amounts to selecting the correct function to continue the drawing based on the geometric type of the object (points, polylines, triangle meshes). Since we are working with a single triangle, this function resolves to `DrawTriMesh`.

3.1.2 BACK-FACE CULLING

The first portion of `DrawTriMesh` is an iteration over the triangles of the mesh. The index array consists of triples of integers (each triple corresponding to a single triangle) and the indices used for a lookup into the array of vertices.

In most circumstances, you want back-face culling to occur. If it is disabled, then you proceed immediately to the clipping stage. If culling is enabled, the default is to eliminate back-facing triangles. However, there are some special effects that require front-facing triangles to be culled. For example, planar reflections need this because the object has counterclockwise-ordered triangles to be back-face culled, but the object's reflection has, effectively, clockwise-ordered triangles. The `CullState` class allows you to select how you want culling to occur, if at all.

Assuming the default is back-face culling, let the triangle vertices be (r'_i, u'_i, d'_i, w'_i) , $0 \leq i \leq 2$, which are the clip-space coordinates of the transformed vertex positions. These are the values mentioned in Section 2.3.5, produced by Equations (2.65) through (2.67). The visibility test you are most likely familiar with is in three dimensions, not in 4D clip space. In three dimensions, if the vertices are $\mathbf{P}_i$ for $0 \leq i \leq 2$ and the eye position is $\mathbf{E}$, the triangle is visible when the triple scalar product is positive,

$$\delta = (\mathbf{E} - \mathbf{P}_0) \cdot (\mathbf{P}_1 - \mathbf{P}_0) \times (\mathbf{P}_2 - \mathbf{P}_0) > 0$$

The cross product portion of this expression produces a normal vector to the plane of the triangle. The eye point must be on the side of the plane to which this normal is directed, a condition captured by $\delta > 0$. If the triangle vertices were instead clockwise ordered, visibility is guaranteed by $\delta < 0$. The sign of the test is represented by the integer `m_iCullSignAdjust` in the software renderer code. Its value is modified to be consistent with the current state of the `CullState` object.

In view coordinates, the eye point is $(r, u, d) = (0, 0, 0)$ since it is the origin of the camera coordinate system. In clip space, the eye point is $(0, 0, -d_{\max}d_{\min}/(d_{\max}, 0))$. The visibility test is naturally three-dimensional, so it is enough to extract three components from the clip-space coordinates to use in computing δ . In fact, these should be the (r', u', w') components. The presence of the offset in the d' component makes (r', u', d') the wrong choice, because the difference of eye point and triangle vertex is not quite the difference you think it is. In the software renderer source code you see the equivalent of

$$\begin{aligned} \mathbf{E}_1 &= (r'_1 - r'_0, u'_1 - u'_0, w'_1 - w'_0), & \mathbf{E}_2 &= (r'_2 - r'_0, u'_2 - u'_0, w'_2 - w'_0), \\ \mathbf{N} &= \mathbf{E}_1 \times \mathbf{E}_2, & \delta_1 &= \sigma (\mathbf{N} \cdot (r'_0, u'_0, w'_0)) \end{aligned}$$

where σ is the sign adjustment based on the cull state (and on the handedness of the camera coordinate system). If $\delta_1 \leq 0$, the triangle is back facing and skipped by the drawing system (i.e., culled). The test may be summarized in the equivalent determinant form as

$$\begin{aligned} \delta_1 &= \sigma \det \begin{bmatrix} r'_0 & u'_0 & w'_0 \\ r'_1 & u'_1 & w'_1 \\ r'_2 & u'_2 & w'_2 \end{bmatrix} \\ &= \sigma (w'_0(r'_1u'_2 - r'_2u'_1) - w'_1(r'_0u'_2 - r'_2u'_0) + w'_2(r'_0u'_1 - r'_1u'_0)) \end{aligned} \quad (3.1)$$

The edge differences could be used for the middle and last row, but the determinant value remains the same.

As always, floating-point round-off errors can cause problems. In the case of back-face culling, the floating-point computations that produce δ_1 can lead to misclassifications. A back-facing triangle might be classified as front facing. This is not a serious problem, because for closed surfaces, the front-facing triangles will overdraw a back-facing triangle. The other misclassification is a problem. When a front-facing triangle is misclassified as back facing, it is not drawn when it should be. This leads to small holes in a rendered scene. Assuming no other triangles are drawn to the same pixels of the missing one, you will see the background color through the holes. In the Wild Magic software renderer, when the visibility test indicates that a triangle is back facing, I do some additional work to check if the triangle is really a misclassified front-facing triangle. My choice is to project the vertices to window coordinates, with truncation to integer values, and to formulate the visibility test in those coordinates. However, you must make sure the vertices are all in front of the camera—this is a matter of verifying that all the w' components are positive. The test without the rounding step is as follows. The projections are

$$(x_i, y_i) = (a_0 + a_1(r'_i/w'_i), b_0 + b_1(u'_i/w'_i)), \quad 0 \leq i \leq 2$$

where a_0 , a_1 , b_0 , and b_1 are constants that depend on the current window width, height, and viewport settings. When the full viewport is in use, the constants are $a_0 = W/2$, $a_1 = W/2$, $b_0 = H/2$, and $b_1 = -H/2$, where W is the window width and H is the window height. The visibility test is

$$\delta_2 = \sigma(x_2 - x_0, y_2 - y_0) \cdot (x_1 - x_0, y_1 - y_0)^\perp \quad (3.2)$$

where $(u, v)^\perp = (v, -u)$. The triangle is back facing when $\delta_2 \leq 0$. It may be shown that the visibility tests are related by $\delta_2 = -a_1 b_1 \delta_1 / (w_0 w_1 w_2)$, where δ_1 is defined in Equation (3.1). Thus, $\text{Sign}(\delta_1) = \text{Sign}(\delta_2)$.

EXERCISE 3.1

For a window of width $W = 640$ and height H , and for $\sigma = 1$, verify that Equations (3.1) and (3.2) produce numerically computed values of the same sign for the following points of the form (r', u', w') :

$$\mathbf{V}_0 = (-4.4538450, -2.6110454, +5.8253970),$$

$$\mathbf{V}_1 = (-4.1239305, -2.5799429, +5.8253970), \text{ and}$$

$$\mathbf{V}_2 = (-4.1239305, -2.6758144, +6.0793653).$$

Project these points to window coordinates and truncate to integer values. Substitute the integer-valued coordinates into Equation (3.2) and show that the result has the opposite sign of that produced by using the floating-point-valued coordinates. ■

- EXERCISE 3.2** Construct a triangle for which Equation (3.1) produces a positive value and Equation (3.2) produces a negative value when using floating-point arithmetic and when the second equation uses the floating-point inputs, not the truncated integer inputs. In this case, the two tests disagree about the orientation of the triangle. Similarly, construct a triangle for which Equation (3.1) produces a negative value and Equation (3.2) produces a positive value when using floating-point arithmetic and when the second equation uses the floating-point inputs, not the truncated integer inputs. ■

3.1.3 CLIPPING

Once a triangle is determined to be front facing, it needs to be clipped against the planes of the view frustum. As mentioned in Section 2.4.3, you can implement clipping in a couple of ways. The first is plane-at-a-time clipping, where a list of triangles is maintained. Initially, the list has one triangle—the one just determined to be front facing. Each triangle is tested for intersection with a frustum plane. If it does intersect, the portion on the frustum side of the plane is placed back in the list for clipping against other frustum planes. If the portion is still a triangle, it is immediately placed back in the list. If the portion is a quadrilateral, it is split into two triangles and these are put back in the list.

The other clipping method was polygon-of-intersection clipping. This method maintains a convex polygon that is initially the front-facing triangle. The polygon is clipped against each of the six frustum planes. The final polygon is then split into a triangle fan, each triangle sent to the rasterizer for drawing. This is the method I choose to implement. The entry point to the clipping system is the function

```
void ClipPolygon (int& riQuantity, int aiIndex[SOFT_MAX_CLIP_INDICES],
                 int aiEdge[SOFT_MAX_CLIP_INDICES]);
```

On input, `riQuantity` is 3, which is the number of vertices of the initial polygon (the front-facing triangle). The `aiIndex` array stores the indices into the vertex array that stores the output from the vertex programs. On output, the value of `riQuantity` is the number of vertices for the convex polygon that was obtained by clipping the triangle against the frustum planes and against user-defined clipping planes. If new vertices were generated by clipping, they are stored in the output vertex array. The first `riQuantity` indices in `aiIndex` represent the vertices of the convex polygon. The array `aiEdge` is used to store information about the edges of the convex polygon but is used only for the purpose of wireframe drawing. When a wireframe is displayed, you want only edges drawn from the original triangles before clipping. The clipper generates new edges, but we do not want these to be displayed. How the information in `aiEdge` supports this distinction will be made clear in a moment.

The `ClipPolygon` function is a shallow wrapper that iterates over the frustum planes and calls a specialized function of the same name but with one additional

parameter—the plane. After the frustum planes are processed, any user-defined clipping planes are processed.

The specialized function is

```
void ClipPolygon (const Vector4f& rkPlane, int& riQuantity,
    int aiIndex[SOFT_MAX_CLIP_INDICES],
    int aiEdge[SOFT_MAX_CLIP_INDICES]);
```

Its job is to test on which side of the frustum plane the vertices lie. Notice that the input plane is passed in as a 4-tuple, $(n_0, n_1, n_2, -d)$, where the plane normal is $\mathbf{N} = (n_0, n_1, n_2)$ and the plane constant is d . A clip-space vertex (r', u', d', w') is on the frustum side of the plane whenever

$$(n_0, n_1, n_2, -d) \cdot (r', u', d', w') \geq 0$$

The function computes p , the number of vertices for which the dot product is positive (vertex is strictly inside the frustum), and n , the number of vertices for which the dot product is negative (vertex is outside the frustum). If k is the number of polygon vertices, then $p + n \leq k$ (vertices exactly on the frustum plane are not counted).

If $p = 0$, then the entire polygon is outside the frustum, possibly with one or more vertices on the frustum plane itself, so the polygon is culled. The returned quantity of vertices is zero so that the calling function knows to terminate clipping; that is, the polygon need not be tested for intersection with any more clipping planes. If $p > 0$ and $n = 0$, then the entire polygon is inside the frustum, possibly with one or two vertices on the frustum plane itself, so the polygon does not need clipping. The last case is $p > 0$ and $n > 0$, so the polygon is partially inside the frustum and partially outside the frustum. It does require clipping and is passed to the function `ClipPolygonAgainstPlane`. The code to compute the dot products also computes the first index for which a dot product is positive. This index is used to start an iteration over the edges of the convex polygon during the clipping phase. The first index is also passed to `ClipPolygonAgainstPlane`.

EXERCISE 3.3

You will notice that the source code compares a dot product δ using $\delta > 0$ and $\delta < 0$. That means the test for a vertex occurring on the frustum plane is the exact comparison $\delta = 0$. Normally, such a comparison is not recommended for floating-point arithmetic, because of numerical round-off errors leading to incorrect classifications. If, instead, a small tolerance $\varepsilon > 0$ were used—say, $\delta > \varepsilon$ to call the dot product positive, $\delta < -\varepsilon$ to call the dot product negative, and $|\delta| \leq \varepsilon$ to call the dot product zero—what are the potential consequences farther down the geometric pipeline? ■

Figure 3.1 shows a convex polygon clipped by an edge of a rectangle that represents the view frustum. For the sake of argument, let the current polygon have k vertices. The polygon of Figure 3.1 has a first vertex inside the frustum, $\mathbf{P}_f$, where $f \geq 0$. The previous vertex is $\mathbf{P}_{f-1}$, where the index is computed modulo k . That is,

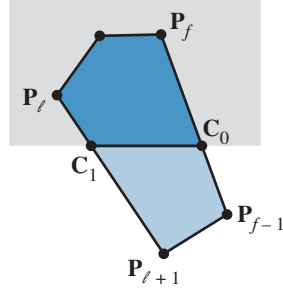


Figure 3.1 A convex polygon clipped by an edge of a rectangle that represents the view frustum.

if $f > 0$, the previous index is $f - 1$. If $f = 0$, the index is $k - 1$ (the same as -1 modulo k). The polygon also has a last vertex inside the frustum, $\mathbf{P}_{\ell}$. Its successor is $\mathbf{P}_{\ell+1}$, where the index is computed modulo k ; if $\ell < k - 1$, then the next index is $\ell + 1$, but if $\ell = k - 1$, then the next index is 0 (the same as k modulo k). The edge from the vertex $\mathbf{P}_{f-1}$ to $\mathbf{P}_f$ intersects the frustum plane, so a clip vertex must be generated:

$$\mathbf{C}_0 = \mathbf{P}_{f-1} + \left(\frac{\delta_{f-1}}{\delta_{f-1} - \delta_f} \right) (\mathbf{P}_f - \mathbf{P}_{f-1})$$

where δ_i is the dot product associated with $\mathbf{P}_i$ relative to the current frustum plane. The positions $\mathbf{P}_i$ are clip-space coordinates that were computed by the vertex program and stored in the output array. Vertex attributes were computed or passed through and also stored in the output array. These must be interpolated as well so that the new clip vertex has its own attributes. The function `ClipInterpolate` does this work and appends the new vertex to the output array. Notice that the interpolation is linear, not perspective. The edge from $\mathbf{P}_{\ell}$ to $\mathbf{P}_{\ell+1}$ intersects the frustum plane, so another clip vertex must be generated:

$$\mathbf{C}_1 = \mathbf{P}_{\ell} + \left(\frac{\delta_{\ell}}{\delta_{\ell} - \delta_{\ell+1}} \right) (\mathbf{P}_{\ell+1} - \mathbf{P}_{\ell})$$

Once again, `ClipInterpolate` has the responsibility for interpolating the vertex positions and attributes.

The clipped convex polygon has vertices

$$\mathbf{C}_0, \mathbf{P}_f, \dots, \mathbf{P}_{\ell}, \mathbf{C}_1$$

The edge from $\mathbf{C}_1$ back to $\mathbf{C}_0$ is the final edge of the polygon, the other edges strictly inside the frustum. The new clip vertices are stored in the output array (by `ClipIn-`

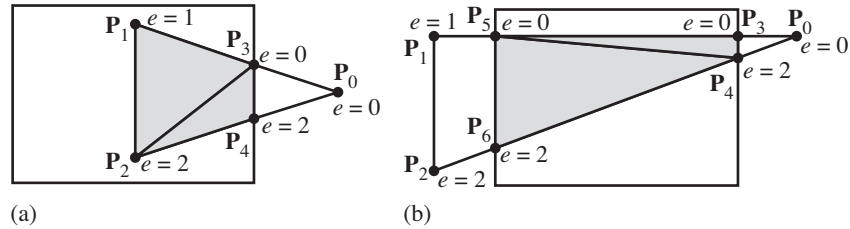


Figure 3.2 (a) A triangle clipped by one edge of a rectangle. (b) A triangle clipped by two edges of a rectangle.

terpolate); call them P_m and P_{m+1} . The polygon indices, in order, are $m, f, \dots, \ell, m+1$. During the clipping, these indices are stored in the temporary array `aiCIndex`. Once clipping is finished, `aiCIndex` is copied to `aiIndex` for use in clipping by the next plane.

Now for an explanation about the input array `aiEdge` to the function `ClipPolygon` and its temporary array counterpart, `aiEIndex`. Figure 3.2 shows two configurations for a triangle clipped by a rectangle that represents the view frustum. Part (a) of the figure shows a triangle clipped by one edge of the rectangle. The original vertices are P_0 , P_1 , and P_2 . The edge index array is initialized with the indices 0, 1, and 2. These are listed in the figure as $e=0$, $e=1$, and $e=2$. When the clip vertex is computed on the edge from P_0 to P_1 , it is assigned the edge index from the first edge vertex. In this case, the clip vertex is P_3 and is assigned $e=0$, which is the edge index for P_0 . The clip vertex for the edge from P_2 to P_0 is P_4 and is assigned $e=2$, which is the edge index for P_2 . The idea is simple—the clip vertex is assigned the index of the original triangle edge on which it lives.

The clipped triangle is a quadrilateral, so it is split into two triangles, $\langle P_1, P_2, P_3 \rangle$ and $\langle P_2, P_4, P_3 \rangle$. The edges $\langle P_2, P_3 \rangle$ and $\langle P_3, P_4 \rangle$ were introduced by clipping, so we do not want them to be displayed in wireframe mode. The `DrawTriMesh` function has a block of code that is entered when wireframe mode is selected:

```
for (int j0 = iQuantity - 1, j1 = 0; j1 < iQuantity; j0 = j1++)
{
    if (aiIndex[j0] < iVQuantity
        || aiIndex[j1] < iVQuantity
        || aiEdge[j0] == aiEdge[j1])
    {
        RasterizeEdge(aiIndex[j0], aiIndex[j1]);
    }
}
```

The value `iVQuantity` is the number of vertices in the output vertex array *before* clipping of any triangles. The value `iQuantity` is the number of vertices of the convex polygon, which is four in our current example. The vertex indices for the convex polygon are stored in `aiIndex`, and the indices of the edges on which these vertices live are stored in `aiEdge`. Edges introduced by the triangle fan are not processed in this loop. In our example, the edge $\langle P_2, P_3 \rangle$ was such an edge. The first two Boolean conditions in the loop allow an edge to be drawn if one of its endpoints is an original vertex of the triangle. In Figure 3.2 (a), the edges $\langle P_3, P_1 \rangle$, $\langle P_1, P_2 \rangle$, and $\langle P_2, P_4 \rangle$ satisfy these conditions. However, the edge $\langle P_4, P_3 \rangle$ does not.

In Figure 3.2 (b), only the edges $\langle P_3, P_5 \rangle$ and $\langle P_4, P_6 \rangle$ should be drawn in wireframe. Neither edge satisfies the condition `aiIndex[j0] < iVQuantity` nor `aiIndex[j1] < iVQuantity`, but they both satisfy the condition `aiEdge[j0] == aiEdge[j1]`.

3.1.4 RASTERIZING

When the triangle fan of the convex polygon is to be drawn, the block of code in `DrawTriMesh` to handle this is

```
int iNumTriangles = iQuantity - 2;
for (int j = 1; j <= iNumTriangles; j++)
{
    RasterizeTriangle(aiIndex[0], aiIndex[j], aiIndex[j+1]);
}
```

where `iQuantity` is the number of vertices in the convex polygon. The function `RasterizeTriangle` has the responsibility to draw the pixels in each of the input triangles, performing perspective interpolation as needed.

The first portion of `RasterizeTriangle` accesses the three clip-space coordinates for the triangle vertices and projects them to window space via the function `ClipToWindow`. The returned values from `ClipToWindow` are `fX` and `fY`, which are the floating-point values for the pixel location; `fDepth`, which is the normalized depth at the pixel (its value is in $[0, 1]$); and `fInverseW`, which is the reciprocal of the w -value of the input clip-space coordinate. The depth and inverse w -values are not needed initially; the floating-point window coordinates are truncated to integer values and are used to start the edge buffer construction.

Before the edge buffer construction, a small block of code is used to trap back-facing triangles. Recall that `DrawTriMesh` itself has code to detect back-facing triangles and cull them, but that block of code uses floating-point arithmetic. Numerical round-off errors can cause triangles that are back facing but nearly front facing (i.e., they are seen nearly edge-on by the camera) to be classified as front facing. These are sent through the clipper and finally to the triangle rasterizer. A more aggressive algorithm could be used to prevent such triangles from reaching the rasterizer, but most likely you would need exact arithmetic to guarantee correct classification all

the time. This is an expense that all triangles would incur, so I chose instead not to penalize the system for all triangles when only a small number have the chance of making their way to the clipper and rasterizer. The rasterizer test for back-facing triangles is an exact test, so any triangles that are rasterized are guaranteed to be front facing.

EXERCISE 3.4 If the back-face culling is removed from the preclipping stage and you allow the aforementioned block of code to do all the back-face culling, will the rendered results be correct? Justify your answer. You can experiment with this by modifying the software renderer and running either the Terrain or VolumeFog samples. ■

3.1.5 EDGE BUFFERS

The edge buffer algorithm is designed to identify the starting and ending x -values on each scan line that contains pixels covered by the triangle. The basic idea is to rasterize the edges connecting the vertices, assigning to each edge pixel the attributes that are obtained by perspective interpolation of the vertex endpoints of the edge. Equation (2.113) is used for the perspective interpolation.

An optimized software renderer will use Bresenham’s algorithm to generate the pixels along an edge. The Wild Magic software renderer is not optimized, using instead floating-point arithmetic to generate the pixels by evaluating the segment $(x_0, y_0) + t((x_1, y_1) - (x_0, y_0))$ connecting the edge endpoints (x_0, y_0) and (x_1, y_1) for appropriate values of t .

Edge rasterizers need to pay attention to processing the pixels along an edge shared by two triangles. First, Bresenham’s algorithm is not guaranteed to produce the same set of pixels when traversing the edge in both directions. That is, if the pixel identification starts with (x_0, y_0) and incrementally generates (x, y) values until it reaches (x_1, y_1) , you can get a different set of pixels if you start with (x_1, y_1) and incrementally generate pixels until you reach (x_0, y_0) . Figure 3.3 shows such an example.

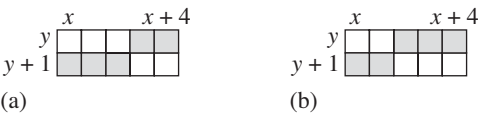


Figure 3.3 Bresenham’s algorithm can generate different sets of pixels depending on the direction of traversal. (a) The pixels are traversed from the left point $(x, y + 1)$ to the right point $(x + 4, y)$. (b) The pixels are traversed from the right point $(x + 4, y)$ to the left point $(x, y + 1)$.

One line-drawing algorithm that avoids the problem of traversal direction is the *midpoint* algorithm. You start with the midpoint (x_m, y_m) of the endpoints and apply Bresenham's algorithm twice, once to iterate from (x_m, y_m) to (x_0, y_0) and once to iterate from (x_m, y_m) to (x_1, y_1) . Regardless of the order in which you process the edge endpoints, you will generate the same sets of pixels.

An alternative, and the one I have implemented, is to guarantee that you always start the pixel iteration from the endpoint of smallest y -value to the endpoint of largest y -value. The function `ComputeEdgeBuffers` arranges for this to happen by sorting the y -values of the triangle vertices. If the three y -values are distinct, two edges are processed to fill in one edge buffer, and the remaining edge is used to fill in the other edge buffer. In the logic of `ComputeEdgeBuffers`, a function `ThreeBuffers` is called to fill the edge buffers. If two y -values are equal, then only two edges need to be processed, one for each edge buffer. A function `TwoBuffers` is called to fill the edge buffers in this case.

Both `ThreeBuffers` and `TwoBuffers` make calls to the function `ComputeEdgeBuffer` to fill in the appropriate buffer. This function assigns the endpoint attributes to the edge buffer, calls the function to perspective interpolate the attributes for the pixels on the edge associated with the edge buffer, and then computes the x -values for those pixels. This information is returned to the triangle rasterizer to be used for perspective interpolation along the scan lines.

Figure 3.4 shows an example of a triangle and the edge buffers associated with it. The minimum edge buffer (on the left) is generated by one edge of the triangle. The maximum edge buffer (on the right) is generated by two edges of the triangle. Both edge buffers have a minimum y -value of 0 and a maximum y -value of 12. The minimum edge buffer has x -values 6, 5, 5, 4, 4, 3, 3, 2, 2, 1, 1, 0, and 0, listed from

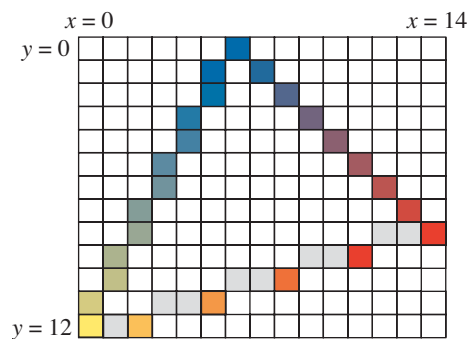


Figure 3.4 A triangle and its associated edge buffers. The minimum edge buffer (on the left) is generated by one edge of the triangle. The maximum edge buffer (on the right) is generated by two edges of the triangle.

minimum y to maximum y . The maximum edge buffer has x -values 6, 7, 8, 9, 10, 11, 12, 13, 14, 11, 8, 5, and 2, listed from minimum y to maximum y . The additional gray squares are those pixels that would be visited using Bresenham's line-drawing algorithm, but they are not part of the edge buffers. Only the pixels with extreme x -values are part of the edge buffers. The vertices of the triangle are colored red, yellow, and blue. The other nongray colors are obtained by interpolation of the vertex colors.

3.1.6 SCAN LINE PROCESSING

Once the edge buffers are constructed for a triangle, we are ready to interpolate the vertex attributes on each scan line intersecting the triangle. The function `RasterizeTriangle` implements the following pseudocode:

```
int ymin = <minimum y for edge buffers>;
int ymax = <maximum y for edge buffers>;
for (y = ymin; y <= ymax; y++)
{
    int xmin = <minimum x for scan line y>;
    int xmax = <maximum x for scan line y>;

    draw the pixel (xmin,y);

    interpolate attributes at (x,y) for xmin < x < xmax;
    for (x = xmin+1; x < xmax; x++)
    {
        draw the pixel (x,y);
    }

    draw the pixel (xmax,y);
}
```

Figure 3.4 shows vertex colors at the edge buffer pixels, but when the vertex attributes are computed during edge buffer construction, the pixels are not actually drawn. The attribute information is stored in addition to the edge buffer pixel locations. The drawing takes place in the aforementioned loops. The rasterization of the triangle in Figure 3.4 is shown in Figure 3.5.

The scan line processing loops have a subtle problem. If two triangles share an edge, the rasterizer will draw the shared pixels twice. Although inefficient, as long as all objects are opaque, the rendered results will be visually correct. However, if any form of blending is used, the results will be incorrect. Figure 3.6 illustrates the problem when the pixel colors generated by the rasterizer are to be added into the current frame buffer. Each pair of triangles has a shared edge. All three triangles share a vertex. As a whole, the three triangles should be drawn as shown in part (b) of the

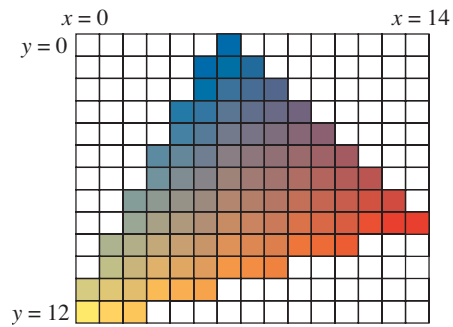


Figure 3.5 The rasterized triangle of Figure 3.4.

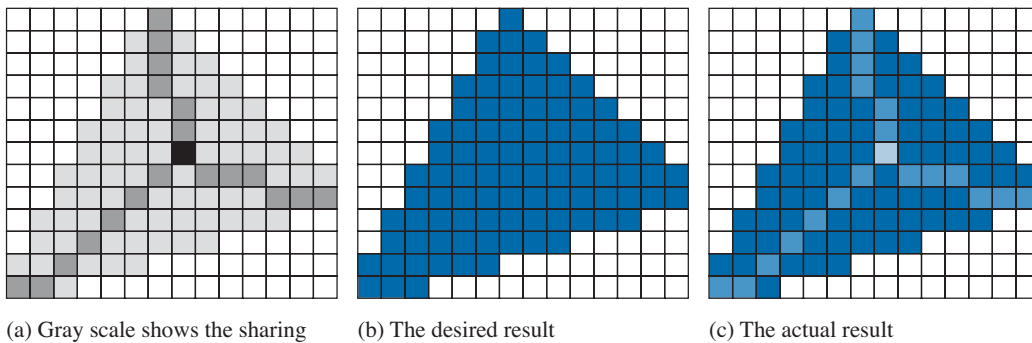


Figure 3.6 Three triangles that share edges. The frame buffer has all black pixels. (a) The pixel sharing is shown in shades of gray. The light gray pixels are not shared at all, each belonging to a single triangle. The dark gray pixels correspond to the edge shared by two adjacent triangles. The black pixel corresponds to a vertex shared by the three triangles. (b) The desired result is to add the triangle pixel colors into the frame buffer for the three triangles as a whole. (c) The actual result when the shared pixels are added multiple times to the frame buffer.

figure. However, they actually are drawn as shown in part (c) of the figure. The vertex color of all the triangle vertices is dark blue. The nonshared pixels have their dark blue color added to the frame buffer. The twice-shared pixels have their colors added twice, producing pixels with a lighter blue color. The pixel shared by all triangles has its color added three times, producing a pixel with the lightest blue color.

Naturally, this example shows it is desirable to draw a mesh of triangles so that pixels corresponding to shared edges and shared vertices are drawn only once. One method for handling this is to rasterize the triangles using the *top-left rule*. A description of this is found in [Cor], but it is missing the details of a pathological case that can arise. The floating-point window coordinates for the triangle vertices are used. During rasterization, the pixel centers (the integer-valued window coordinates) are analyzed for containment in the triangle. If a pixel center is strictly inside the current triangle, then the pixel is drawn by the rasterizer. If the pixel center is outside the current triangle, then the pixel is not drawn. If an adjacent triangle contains the pixel center in its interior, the pixel will be drawn when that triangle is rasterized. The difficult case is when a pixel center is on an edge of the triangle. If the edge is shared by two triangles, a decision must be made regarding which triangle should be responsible for drawing each of the pixels that correspond to the edge.

Let the triangle have vertices (x_i, y_i) for $0 \leq i \leq 2$, which are floating-point window coordinates. A *top edge* of a triangle is one for which two vertices have the same y -value and the other vertex has a y -value that is larger. For example, if $y_0 = y_1 < y_2$, the triangle has a top edge. A *left edge* of a triangle is one that is to the left of the triangle interior in the x -direction. For example, if $y_0 < y_1 < y_2$ and $x_1 < x_0 < x_2$, the edge from (x_0, y_0) to (x_1, y_1) is a left edge and the edge from (x_1, y_1) to (x_2, y_2) is a left edge. Another example using the same y -values is $x_0 < x_2 < x_1$. The edge from (x_0, y_0) to (x_2, y_2) is a left edge.

When a pixel center lies exactly on a triangle edge, the top-left rule is used as a tie-breaker. If the pixel center lies on the top edge of the current triangle, this triangle is responsible for drawing the pixel. An adjacent triangle sharing the edge has a *bottom edge* but ignores pixels whose centers are exactly on this edge. Similarly, if the pixel center lies on the left edge of the current triangle, this triangle is responsible for drawing the pixel. An adjacent triangle sharing the edge has a *right edge* but ignores pixels whose centers are exactly on this edge.

EXERCISE 3.5

Consider two triangles whose vertices are listed here in floating-point window coordinates. The origin of the window is the upper-left corner, the x -values increase rightward, and the y -values increase downward. The first triangle has vertices $(10, 1)$, $(5, 10)$, and $(10, 10)$. The second triangle has vertices $(10, 1)$, $(10, 10)$, and $(15, 10)$. Each triangle has a bottom edge but not a top edge. The pixel center $(10, 1)$ lies on the edges of both triangles. The pixel center is on the left edges for both triangles. Which triangle is assigned the responsibility for drawing the pixel? (This is the pathological case missing in the description in [Cor].) ■

The Wild Magic software renderer uses a simpler mechanism for avoiding the multiple drawing of shared pixels in a mesh of triangles. The rasterizer draws only those pixels (x, y) for which $y_{\min} \leq y < y_{\max}$ and $x_{\min}(y) \leq x < x_{\max}(y)$, where the x -bounds are the edge buffer values. That is, the last row and last column in which triangle pixels occur are ignored. If an adjacent triangle shares pixels in the last row (last column) and those pixels are not in the last row (last column) of the

adjacent triangle, that triangle will draw the pixels. Omitting the upper bounds is a trivial modification to the edge buffering mechanism. In exchange, the rasterizer never draws the bottommost and rightmost nonshared pixels in a triangle mesh. The rendered scenes still look reasonable; in fact, my experiments show me that they actually resemble the Direct3D and OpenGL renderings more so than when using schemes that fail to omit those pixels. The major visual problem with this approach, though, is that for a rendering that fills the entire window, you lose the pixels on the last row and the last column of the window—the background color appears instead. To avoid this, I modified the edge buffers to store $W + 1$ rows, where W is the window width. The transformation from clip space to window space clamps x to $[0, W]$ and y to $[0, H]$, where H is the height of the window. The last rows of the edge buffers are computed when a scene fills the window, but pixels of the form (W, y) or (x, H) are never sent to the pixel program, because of the restrictions imposed in the rasterizer: $x < x_{\max}$ and $y < y_{\max}$. The `RasterizePoint` and `RasterizeEdge` functions do not use the edge buffering system. The window coordinates for points and edge vertices can be out of range, so these two functions clamp x to $[0, W - 1]$ and y to $[0, H - 1]$.

EXERCISE 3.6

The function `RasterizeEdges` is used for wireframe mode. The source code does not currently avoid multiple drawing of shared pixels. The potentially shared pixels are the endpoints of the edges. Modify the source code to avoid the overdraw. You may use the `SampleGraphics/Lighting` to test this. Run the sample as is. Press the A-key to add an ambient light and press the D-key to add a directional light. Toggle to wireframe mode by pressing the w-key. You will see that the vertices are brighter than others because of the overdraw during addition of the two light contributions. When you have the buffer support added to the wireframe drawing, you should no longer see this effect. ■

3.1.7 PIXEL SHADERS

In `RasterizeTriangle`, when a pixel is finally determined to be drawable, a call is made to the function `ApplyPixelShader`. Without concern for depth buffering, stencil buffering, blending, or color masking, this function has the role of applying a *pixel shader* for the current pixel. This amounts to calling a *pixel program* whose inputs are (1) the rasterized and interpolated outputs from the vertex program and (2) any constants that are needed to obtain a desired effect. The pixel program must return an RGBA color value.

Previously, we saw a Cg vertex program for vertex coloring. The output of the vertex program was just the input vertex color. A Cg pixel program that just passes through the color is listed next. You can find this on the CD-ROM together with other Cg shader programs in the directory

```
GeometricTools\WildMagic4\Data\ShaderPrograms\Cg
```

The file of interest is `PassThrough.cg`. A pixel program in that file is

```

void PassThrough3PProgram
(
    in float3 kInPixelColor : COLOR,
    out float4 kPixelColor : COLOR)
{
    kPixelColor.rgb = kInPixelColor;
    kPixelColor.a = 1.0f;
}

```

The program has one input. This is the color produced by the rasterizer for the current pixel and is an interpolation of the vertex colors at the triangle vertices. The vertex program `VertexColor3VProgram` passed through only RGB values. The output of the pixel program must assign an alpha value, in this case 1 to produce an opaque color.

The software version of the pixel program returns the RGBA color as the result of the function. The actual line of code in `ApplyPixelShader` that calls the pixel program is

```
ColorRGBA kSrcColor = m_oPProgram(afRegister,apkSampler,afInPixel);
```

The first parameter is an array of constants. In our current example, there are no constants, so the pixel program does not dereference the array pointer. The second parameter is an array of *texture samplers* to be used for color lookups in texture images. The current program does not use textures, so this array is not dereferenced. Texture samplers are discussed later in this section. The last parameter is the array of interpolated outputs from the vertex program, not including the clip-space position. Each float channel of `afInPixel` is an interpolation of the corresponding channel of the three vertices of the triangle.

It is possible that the color returned from the pixel program has channels outside the interval $[0, 1]$. Eventually, the colors are mapped to 8-bit quantities for the frame buffer, so they need to be clamped first. The relevant software renderer code for this is condensed to

```

ColorRGBA kSrcColor = m_oPProgram(afRegister,apkSampler,afInPixel);
kSrcColor.Clamp(); // clamp to [0,1]
SetColorBuffer(iIndex,kSrcColor);

```

The variable `iIndex` is the index into the color buffer for the current pixel (x, y) . If W is the window width, the index is $i = x + Wy$. The color buffer elements are stored in row-major order. The function `SetColorBuffer` draws the color in the frame buffer (or render target). This operation is hidden from the user because the storage format for the RGBA colors is platform-specific; PCs use little endian, and Macintoshes (those before the switch to Intel CPUs) use big endian.

As it turns out, the implementation of `ApplyPixelShader` is more complicated than the few lines of code shown here. Depth buffering, blending, and stencil buffering must be incorporated into the logic of when and how to draw a pixel. In Direct3D, the pixel pipeline involves the following ordered list of operations:

- *Pixel shader*. The pixel program is called first, regardless of whether the result is actually used in the display. That is, the pixel may be rejected by various tests in the pipeline.
- *Occlusion queries*. The occlusion queries keep track of the number of pixels that pass z-testing. This mechanism may be used only when you draw the objects from front to back. I do not include this feature in the software renderer.
- *Scissor test*. The scissor test culls pixels outside a specified subrectangle of the render target. The software renderer uses this concept for clearing buffers but does not support the actual culling in the `ApplyPixelShader` function call.
- *Alpha test*. The pixel may be rejected for further processing based on a comparison of its alpha value to a reference value. The `AlphaState` object has support for this, but the software renderer does not yet implement it.
- *Stencil test*. A stencil buffer may be used to support various special effects such as projected shadows, planar reflections, shadow volumes, and compositing. The software renderer does have support for this. A pixel may be rejected for further processing depending on the state of the stencil buffer.
- *Depth test*. A pixel is determined to be visible (or not) based on its depth value relative to a value stored in a depth buffer. The comparison function is user selectable. The depth buffer can be configured for read-only, write-only, or read-write.
- *Fog effects*. A fog factor may be computed to blend the pixel color (returned by the pixel shader) and a fog color. The software renderer does not include this stage in its pixel pipeline. It is simple enough to support fog effects within the pixel shader itself.
- *Alpha blending*. The pixel color returned by the pixel shader may be blended with the current render target color at that pixel's location. Control over the blending factors is provided by the `AlphaState` object.
- *Dithering*. This was a common method to give the impression that the device supports more colors than physically possible, relying on the human visual system to blend colors it sees. For example, if you juxtapose a red pixel and a blue pixel, your eye would blend these and see a magenta pixel. Given the powerful graphics cards we have these days, the rendering system in Wild Magic does not expose the ability to dither, even if the hardware supports it.
- *Color masking*. Some special effects are possible that require using only some of the color channels of the pixel color. The channels may be masked as needed to guarantee that only the desired channels are read and/or written.

- *Display.* The final pixel color is written to the render target, typically the frame buffer, but often an offscreen chunk of memory.

As mentioned, the software renderer does not currently support all these features. Moreover, the order of the operations is different from what Direct3D and OpenGL support. The main reason for this is that the pixel shader is the bottleneck in a software rendering system. Most graphics systems are fill-rate limited; it takes a lot of time to process a large number of pixels. Rather than always calling the pixel program at the beginning of the pipeline, the software renderer waits to see if the pixel passes the stencil and depth tests (and the color masking test—there’s nothing to draw if all channels are masked out). If it does, then the pixel program is called and, if enabled, alpha blending is performed between the returned color and the corresponding color in the render target’s color buffer. The pipeline currently is

stencil test → depth test → alpha blending → color masking → display

EXERCISE 3.7 In the software renderer, implement the following features: scissor test, alpha testing, fog effects. ■

3.1.8 STENCIL BUFFERING

A stencil buffer gives you the ability to store and retrieve information about pixels that have been visited during rasterization. A pixel is rejected for further processing based on a comparison between a reference value and the pixel’s corresponding stencil buffer value. The first block of code in `ApplyPixelShader` is

```
if (m_auStencilBuffer)
{
    StencilState* pkStencilState = GetStencilState();
    if (pkStencilState->Enabled)
    {
        bool bStencilPass = ApplyStencilCompare(iIndex);
        if (!bStencilPass)
        {
            ApplyStencilOperation(iIndex, pkStencilState->OnFail);
            return;
        }
    }
}
```

If a stencil buffer exists and is enabled, a comparison is made between the reference value and stencil value by calling `ApplyStencilCompare`. The comparison expression is abstractly written as

```
(reference AND mask) comparison_function (stencil_value AND mask);
```

The comparison function is one of never, always, equal, not equal, less than, less than or equal, greater than, or greater than or equal. If the comparison is never, the comparison expression is false regardless of the values of any variable in the expression. If the comparison is always, the comparison expression is always true. The Boolean value of the expression for the other comparisons is clear. The mask value is part of the stencil state and is a mask for reading the stencil values.

If the comparison expression is false, the stencil buffer is modified by calling `ApplyStencilOperation`. The modification occurs according to the stencil operation stored by the `OnFail` member. The operation is one of the following:

- zero. Set the stencil buffer value to zero.
- keep. Keep the current stencil buffer value; that is, the stencil buffer is unchanged.
- replace. Replace the stencil buffer value by the reference value.
- invert. Invert the bits of the stencil buffer value; that is, a 1-bit is changed to a 0-bit and a 0-bit is changed to a 1-bit.
- increment. The stencil value is incremented by 1 as long as its current value is smaller than the maximum stencil value.
- decrement. The stencil value is decremented by 1 as long as its current value is positive.

The modification to the stencil buffer is

```
oldv = stencil[i];
newv = <value computed based on the stencil operation>;
stencil[i] = (oldv AND (NOT wMask)) OR (newv AND wMask);
```

where `wMask` is part of the stencil state and is a mask for writing.

If the stencil comparison is favorable, the pixel is passed to the next stage in the pixel pipeline. In the software renderer, this stage is the depth test, which also involves a comparison called the *z-test*. If the pixel fails the *z-test* and stencil buffering is enabled, the stencil buffer is modified by calling `ApplyStencilOperation`. The modification occurs according to the stencil operation stored by the `OnZFail` member. If the pixel passes the *z-test*, the stencil buffer modification occurs according to the stencil operation stored by the `OnZPass` member.

The read and write masks, the reference value, and the stencil operations give you a lot of power to generate interesting special effects. A couple of sample applications illustrate this. In the `SampleGraphics` folder, look at the `PlanarShadows` and `PlanarReflections` samples. In particular, the functions `PlanarShadows::Draw` and `PlanarReflections::Draw` use the stencil buffer and contain comments about how it is used.

3.1.9 DEPTH BUFFERING

The depth buffer supports occlusion on a per-pixel basis. The state information is stored in the ZBufferState object. Just like stencil state, there is a comparison function used to decide if the current pixel is rejected from further processing. The depths are the inputs to the comparison. The standard comparison is that a pixel is rejected when its depth is larger than that of any earlier drawn pixels at that same pixel location. However, some special effects use other comparisons.

The block of code in ApplyPixelShader that does the depth test is

```
if (m_auiDepthBuffer)
{
    ZBufferState* pkZBufferState = GetZBufferState();
    unsigned int uiDepth =
        WM4_UNIT_FLOAT_TO_UINT(fDepth,m_uiMaxDepthValue);

    StencilState* pkStencilState = GetStencilState();
    if (pkZBufferState->Enabled)
    {
        bool bZPass = ApplyZBufferCompare(iIndex,uiDepth);
        if (!bZPass)
        {
            if (pkStencilState->Enabled)
            {
                ApplyStencilOperation(iIndex,pkStencilState->OnZFail);
            }
            return;
        }
    }

    if (pkZBufferState->Writable)
    {
        m_auiDepthBuffer[iIndex] = uiDepth;
        m_afDepthBuffer[iIndex] = fDepth;
    }

    if (pkStencilState->Enabled)
    {
        ApplyStencilOperation(iIndex,pkStencilState->OnZPass);
    }
}
```

The stencil buffer handling is interleaved with this code. The depth testing logic is to compute the integer-valued depth for the current pixel and compare it to the current

value in the depth buffer. If the comparison is unfavorable, the pixel is rejected for further processing (except for the modifications to the stencil buffer, if enabled). If the comparison is favorable, the depth buffer is updated as long as it is writable. Some special effects use the depth buffer as read-only, others use it as write-only, but the typical use is read-and-write.

3.1.10 ALPHA BLENDING

Once the pixel passes the stencil test (if enabled) and the depth buffer test (if enabled), it is ready to be drawn. The pixel program is called to produce the *source color* for the pixel. The current color buffer value is called the *destination color*. If blending is disabled, the source color is written to the color buffer. If blending is enabled, the source color and destination color are combined according to blending parameters specified in the AlphaState object. These parameters control how Equation (2.108) is interpreted. The parameters themselves are described in Tables 2.5 and 2.6.

A portion of the alpha blending code in ApplyPixelShader is

```
ColorRGBA kSrcColor = m_oPProgram(afRegister,apkSampler,afInPixel);
kSrcColor.Clamp();

AlphaState* pkAlphaState = GetAlphaState();
if (!pkAlphaState->BlendEnabled)
{
    // No blending, so write the source color to the color buffer.
    SetColorBuffer(iIndex,kSrcColor);
    return;
}

// The current color buffer value.
ColorRGBA kDstColor = m_akColorBuffer[iIndex];

// The final color after blending.
ColorRGBA kFinalColor = GetBlendedColor(kSrcColor,kDstColor);
kFinalColor.Clamp();

SetColorBuffer(iIndex,kFinalColor);
```

When blending is disabled, the first call to SetColorBuffer writes the source color, kSrcColor, to the color buffer for display. If blending is enabled, the destination color, kDstColor, is read from the color buffer. The two colors are passed to GetBlendedColor to be combined according to the state in the current AlphaState object. The final color is returned, clamped, and written to the color buffer.

3.1.11 COLOR MASKING

The alpha blending code is inside a condition statement

```
if (m_auiColorBuffer && m_uiColorMask != 0)
{
    // . . . alpha blending code . . .
}
```

As long as there is a color buffer and not everything is masked away, the alpha blending code is called. The color mask and the reads and writes to the color buffer are dependent on the platform, specifically on the byte ordering used to store RGBA colors. An application can set the mask via the virtual function `SetColorMask`. This function is implemented in the renderer for each specific platform. Writing to the color buffer is implemented in the virtual function `SetColorBuffer` for each derived-class renderer. The color blending is done using the platform-independent class `ColorRGBA`. The destination color is read from an auxiliary buffer that stores the color buffers using `ColorRGBA` values. This avoids loss of precision when blending. The channels of `ColorRGBA` are floating-point numbers in the interval $[0, 1]$. When a color is written to the color buffer, the channels are converted to 8-bit quantities, which is a loss of precision. If you were to read back the 32-bit color buffer, convert to a `ColorRGBA`, and blend it with the source color, the final result would not be the same as if you had blended the two original `ColorRGBA` values.

The function `SetColorBuffer` uses the color mask, `m_uiColorMask`, to correctly access the channels of the color buffer for updating.

3.1.12 TEXTURE SAMPLING

When the triangle mesh uses textures, the vertices are assigned texture coordinates. These must be interpolated by the rasterizer and passed to the pixel program to be used as lookups into texture images. The process of determining the color associated with a texture coordinate is called *texture sampling*.

Section 2.6.3 has a fairly detailed description of how the lookups occur, including how texture coordinates are interpreted (clamp, repeat, and so on), how images are filtered (nearest, linear), and how mipmaps are computed and sampled (combinations of nearest and linear, and anisotropic filtering).

The texture sampling system in the Wild Magic software renderer is comprised of the classes whose abstract base class is `SoftSampler`. The system provides a factory for creating a `SoftSampler` object from a `Texture` object. The system also encapsulates mipmap creation and mipmap selection. The base class implements the function `GetImageCoordinate`, whose job it is to take a channel of a texture coordinate and modify it according to the wrap mode (clamp, clamp to edge, clamp to border,

repeat, or mirrored repeat). The returned value is the modification. The sampling itself depends on the type of sampler. The base class specifies the pure virtual function

```
virtual ColorRGBA operator() (const float* afCoord) = 0;
```

Each derived class implements this according to its needs. A pixel program that uses textures receives an array of samplers. A texture coordinate is passed to a sampler using `operator()` and the returned value is the color to be used by the pixel program. For a simple base texture effect, the returned color is the pixel color.

The derived classes `SoftSampler1`, `SoftSampler2`, and `SoftSampler3` implement texture sampling for 1-, 2-, and 3-dimensional textures, respectively. The derived class `SoftSamplerCube` implements sampling for a cube map, a collection of six images that form a cube and are used for environment mapping. The derived class `SoftSamplerProj` implements sampling for a projected texture, which I require to manage a 2D image. The texture coordinate is of the form (s, t, r, q) , a homogeneous point. The actual texture coordinates for the 2D image are $(s/q, t/q)$, where the perspective division occurs just as for homogeneous points used for vertices of a geometric primitive. If the projected texture is created as a *depth texture* for effects such as *shadow maps*, the value r/q represents the distance from the projector source (e.g., a light to cast a shadow) to the point on the shadow caster. This may be compared to the gray-scale color corresponding to the depth texture sampled at $(s/q, t/q)$ to decide if the pixel corresponding to the point is shadowed or not.

3.1.13 FRAME BUFFERS

The software renderer has quite a collection of buffers that are used to control the final results of the rendering. The color buffer stores the actual pixel colors; the depth buffer stores depth for per-pixel occlusion culling; and the stencil buffer is used for interesting special effects. Although a class such as `SoftRenderer` could have data members for each of these, the buffers are encapsulated by a class `SoftFrameBuffer`. The base class is `FrameBuffer`, which is a class defined in the platform-independent graphics library.

By encapsulating the buffers into a frame buffer, we may easily support the standard frame buffer corresponding to the displayable window, but we may also support *offscreen rendering*. It is very convenient to render a scene to a chunk of memory that is intended for use as a 2D texture to be applied to another object in the scene. This operation is called *render-to-texture* and the offscreen frame buffer is called a *render target*.

The folder `SampleGraphics` has a couple of sample applications that illustrate the use of offscreen rendering. The application `RenderToTexture` displays a triangle mesh in the main window but also the same mesh rendered to a texture that is used for a screen-space polygon occurring in the lower-left corner of the window. The application also renders the mesh to a depth texture. A pseudocolored image is built

from the depth texture and displayed in the lower-right corner of the window. The projector used for this is *not* the camera; rather, it is a source located to the right of the triangle mesh as you are looking at it on the screen. The sample ShadowMaps is a much more complicated use of offscreen rendering.

3.2 HARDWARE RENDERING

Naturally, you purchased this book because of an interest in real-time graphics. As a user of current-generation 3D graphics hardware, most of the rendering system is hidden from you by graphics APIs such as OpenGL or Direct3D. These APIs require you to make various function calls to configure the renderer to draw objects with all the special effects you have in mind. It is not really necessary to know all the details of the internal rendering system, details such as the ones I just presented in the previous section on software rendering.

Ignoring the shader programming aspects of GPUs, the functions available to you are labeled in total as the *fixed-function pipeline*. Much of the render-state management is encapsulated by the API calls, allowing you to focus on the higher-level details of your graphics system. In exchange, though, figuring out *which* functions to call and *in what order* is your responsibility. The multitude of downloadable SDKs, tutorials, and samples, especially from graphics card manufacturers' websites, lessens the responsibility to some extent, but as evidenced by many of the questions posted to game developer forums and Usenet newsgroups, the fixed-function pipeline is still misunderstood by many newcomers to the field of graphics. Although you do not have to know most of the internal details involved in the rendering process, studying the implementation of a software renderer might very well give you insight as to *why* you call the API functions that you do. The Wild Magic software renderer is a simple implementation. For a much richer renderer with more details than you could ever imagine, see Brian Paul's Mesa 3D Graphics Library [Pau06]. This is a software implementation that is intended to be very similar to OpenGL. The source code is regularly maintained.

With the introduction of programmable GPUs, we as graphics developers have been given greater power to invent and display sophisticated special effects, accomplished via shader programming. As is always the case, with great power comes great responsibility. The encapsulation of state management by the fixed-function pipeline is now generally your responsibility. For example, setting up light parameters for dynamic lighting used to be a matter of calling a handful of graphics API functions. Now you must pass them to shader programs and implement the lighting equations yourself. On the other hand, you are no longer obligated to use the lighting model that the fixed-function pipeline uses. As another example, enabling texture units for a single-pass multitexture effect is quite generic in the fixed-function pipeline. Now you must write shader programs for each type of single-pass multitexture effect of interest. Because the number of possible combinations is quite large, it would be

fruitless to manually create all these. For example, if you have four light types (ambient, directional, point, and spot), the number of shader programs to support any combination of these with at most eight lights is 494.¹ For a graphics system to support a very large number of effects through shader programs, it is helpful to have automatic generation of shader programs and to have *shader stitching*—the ability to take preexisting shader programs, combine them into a single program, and compile it on the fly. The alternative is to use multipass programming, but this is expensive since each additional pass requires a re-rasterization of the triangles in the mesh.

To have good performance, the graphics system should also take advantage of the video memory (VRAM) provided by the graphics card. Anything that can be sent to VRAM once and cached for later use should be, whether it is vertex data or texture images. Data that changes infrequently can be stored in *accelerated graphics port* (AGP) memory to take advantage of the faster transfer from AGP memory to VRAM as compared to the transfer from system memory to VRAM.

The rendering layer in Wild Magic version 3 and earlier was an attempt to make the abstract renderer API look like the fixed-function pipeline. It provided some function calls that implemented special effects such as bump-mapping, environment mapping, and projected textures. Any time you created a new and complicated effect, particularly one that required multipass rendering, you had to modify the *Renderer* class by adding a pure virtual function for that effect and implementing it in the derived-class renderers for OpenGL and Direct3D. Wild Magic version 3 also supported shader programs, but by requiring you to generate C++ classes, each encapsulating a pair of vertex and pixel programs. Once again, this was an attempt to make the renderer look like it was a fixed-function pipeline. Some performance enhancements were added, namely, the ability to cache vertex buffers in VRAM, but the scene graph management system and the Direct3D renderer were not structured to be friendly to such a system. The OpenGL renderer performed well, but the Direct3D renderer did not.

The Wild Magic version 4 rendering system is a complete rewrite of the earlier systems. It is designed to be a *resource management system* for the underlying graphics system (hardware or software). The resources include vertex buffers (storage of vertex positions and attributes), index buffers (storage of indices for mesh topology), vertex and pixel programs, and textures and associated texture coordinates and images. The system also includes support for clearing various buffers (color, depth, stencil). The renderer manages a camera, dynamic lights, transformations for objects, and global render states such as depth buffering and alpha blending. Other functions in the system support various items such as text rendering, font selection, user-defined clip planes, color masking, and depth range.

1. If you have an alphabet with n distinct letters and you want to form distinct words of exactly length ℓ , the number of such words is $\text{Choose}(n + \ell - 1, \ell) = (n + \ell - 1)! / (\ell!(n - 1)!)$. The number of distinct words with length at most ℓ is $\sum_{i=1}^{\ell} \text{Choose}(n + i - 1, i)$.

A large set of standard shader constants are supported, including various matrix transformations, light and material parameters, camera parameters, and projector parameters. Functions to set these are not part of the public interface, but are used in the setup for enabling vertex and pixel programs.

The drawing system is the heart of the renderer and is the main client of the resource handling. The next section describes the abstract rendering API for Wild Magic 4, including the role that subsystems play in applying shader programs to geometric primitives.

3.3 AN ABSTRACT RENDERING API

This section describes the abstract base class `Renderer`, which acts as a resource manager for drawing objects. The interface is quite large, so instead of showing a code block of the entire class structure, I will list the interface a small portion at a time and describe the role of each portion.

3.3.1 CONSTRUCTION AND DESTRUCTION

The base class `Renderer` is abstract because its only constructor is protected and because it specifies quite a few pure virtual functions that must be implemented by derived classes. Function members relevant to construction, destruction, and accessing information passed to the constructor are listed next.

```
Renderer (
    FrameBuffer::FormatType eFormat,
    FrameBuffer::DepthType eDepth,
    FrameBuffer::StencilType eStencil,
    FrameBuffer::BufferingType eBuffering,
    FrameBuffer::MultisamplingType eMultisampling,
    int iWidth, int iHeight);

virtual ~Renderer ();

FrameBuffer::FormatType GetFormatType () const;
FrameBuffer::DepthType GetDepthType () const;
FrameBuffer::StencilType GetStencilType () const;
FrameBuffer::BufferingType GetBufferingType () const;
FrameBuffer::MultisamplingType GetMultisamplingType () const;
int GetWidth () const;
int GetHeight () const;
```

```
enum
{
    OPENG,
    DIRECTX,
    SOFTWARE,
    MAX_RENDERER_TYPES
};
virtual int GetType () const = 0;
```

The constructor accepts inputs that describe what frame buffer components are required by the application. The format is typically 32-bit RGBA or 24-bit RGB. The depth type specifies the number of bits for the depth buffer, currently either none, 16, 24, or 32. If you plan on creating derived classes for graphics systems supporting a different number of bits (e.g., SGI O2 machines have a 15-bit format), then you must add new constants to the `FrameBuffer::DepthType` enumeration. The stencil type specifies the number of bits for the stencil buffer, currently none or 8. The buffering type is either single or double. For real-time applications, you want double buffering. The multisampling type refers to the ability to generate multiple samples per pixel during rasterization. Your options for number of samples are none, 2, or 4. The width and height of the buffers are also specified at construction time.

Member accessors are provided for the width, height, and the frame buffer parameters. Also listed here is the function `GetType`, a simple form of run-time type information (RTTI). The only derived classes currently supported by the engine are `OpenGLRenderer` (enumeration `OPENG`), `Dx9Renderer` (enumeration `DIRECTX`), and `SoftRenderer` (enumeration `SOFTWARE`). The OpenGL and software renderers run on platforms other than those with Microsoft Windows. Each platform derives classes from these to provide for a small amount of platform-dependent behavior. For example, the Microsoft Windows OpenGL renderer is class `WglRenderer`, which is derived from `OpenGLRenderer`. The abstract rendering API does not need to know about this level of derivation, only that the renderer is based on OpenGL; thus, there are no enumerations other than those listed here.

3.3.2 CAMERA MANAGEMENT

To draw anything, the renderer needs a camera in order to define the region of space to be rendered and to define the viewing model (perspective or orthographic). The relevant interface for the camera is

```
void SetCamera (Camera* pkCamera);
Camera* GetCamera () const;
virtual void OnFrameChange ();
virtual void OnFrustumChange ();
virtual void OnViewportChange () = 0;
```

The function `SetCamera` tells the renderer to use the specified camera for drawing. This call establishes two-way communication between the renderer and the camera—the renderer has a pointer to the camera and the camera in turn has a pointer to the renderer. When the camera is moved, reoriented, or has its frustum parameters changed, the renderer is automatically notified via the functions `OnFrameChange`, `OnFrustumChange`, and `OnViewportChange`. In *Wild Magic 3*, the derived-class renderers had to implement all three of these, making calls to the graphics APIs to update the homogeneous matrices associated with the camera coordinate system, camera viewing model, and viewport. The observations made in Section 2.8 allow me to set the matrices the way I want them, not the way a graphics API sets them via its convenience functions, so the matrix storage and much of the matrix handling was factored out of the derived classes and placed in the base class. The world matrix, view matrix, and projection matrix are maintained explicitly by `Renderer`. The derived classes have a small amount of work to do, namely, to tell the graphics APIs what their matrices should be.

3.3.3 GLOBAL-STATE MANAGEMENT

The renderer maintains a list of the currently active global states for alpha blending, culling (how to cull triangles), fog, materials, polygon offset (how to deal with depth aliasing problems), stenciling, wireframe, and depth buffering. The accessors are

```
virtual void SetAlphaState (AlphaState* pkState);
virtual void SetCullState (CullState* pkState);
virtual void SetFogState (FogState* pkState);
virtual void SetMaterialState (MaterialState* pkState);
virtual void SetPolygonOffsetState (PolygonOffsetState* pkState);
virtual void SetStencilState (StencilState* pkState);
virtual void SetWireframeState (WireframeState* pkState);
virtual void SetZBufferState (ZBufferState* pkState);
AlphaState* GetAlphaState ();
CullState* GetCullState ();
FogState* GetFogState ();
MaterialState* GetMaterialState ();
PolygonOffsetState* GetPolygonOffsetState ();
StencilState* GetStencilState ();
WireframeState* GetWireframeState ();
ZBufferState* GetZBufferState ();
void SetReverseCullFace (bool bReverseCullFace);
bool GetReverseCullFace () const;

void SetGlobalState (GlobalStatePtr aspKState[]);
void RestoreGlobalState (GlobalStatePtr aspKState[]);
```

```
void SetLight (int i, Light* pkLight);
Light* GetLight (int i);
```

The Set/Get functions for the global states are straightforward, accessing an array of pointers to the currently active states. The functions SetReverseCullFace and GetReverseCullFace are used for special effects involving mirroring, whereby the triangle vertex order of the mirrored image is the reverse of that for a normal image.

The functions SetGlobalState and GetGlobalState are for internal use by the drawing system. These are used on a per-object basis when drawing. In particular, if an object itself has global states attached to it, these override the currently active global states during drawing, but after drawing the previous global states are restored.

The SetLight and GetLight functions are simple accessors for an array of pointers to Light objects. These functions are also for internal use when drawing objects that use lights in support of special effects.

3.3.4 BUFFER CLEARING

Before drawing a collection of objects, the various buffers making up the frame buffer might have to be cleared. The relevant interface is listed here.

```
virtual void SetBackgroundColor (const ColorRGBA& rkColor);
const ColorRGBA& GetBackgroundColor () const;

virtual void ClearBackBuffer () = 0;
virtual void ClearZBuffer () = 0;
virtual void ClearStencilBuffer () = 0;
virtual void ClearBuffers () = 0;
virtual void DisplayBackBuffer () = 0;

virtual void ClearBackBuffer (int iXPos, int iYPos, int iWidth,
                             int iHeight) = 0;
virtual void ClearZBuffer (int iXPos, int iYPos, int iWidth,
                          int iHeight) = 0;
virtual void ClearStencilBuffer (int iXPos, int iYPos, int iWidth,
                                int iHeight) = 0;
virtual void ClearBuffers (int iXPos, int iYPos, int iWidth,
                          int iHeight) = 0;
```

Typically, all buffers are cleared. Double-buffered rendering amounts to rendering the scene to a back buffer (a color buffer), followed by a swap to the front buffer (the color buffer displayed on the screen). The swap occurs through a call to DisplayBackBuffer. The functions named ClearBackBuffer clear the back buffer by setting all its values to the current background color. You may set the background color using SetBackgroundColor. The depth buffer is cleared by the functions named

ClearZBuffer, and the stencil buffer is cleared by the functions ClearStencilBuffer. All three buffers are cleared by the functions named ClearBuffers. The clear functions in the first block are used to clear the entire buffer, but the clear functions in the second block are used to clear a subrectangle of the buffer.

The typical order of operations is

```
clear all buffers;
render the scene;
display back buffer;
```

However, this is not required. For example, if you have a sky dome whose contents fill the window, and then the scene objects are rendered after the sky dome is rendered, there is no need to clear the back buffer. You might also have a level with pre-rendered static geometry that produces both a color buffer and a depth buffer. The intent is that these buffers are loaded from disk and written to the hardware via graphics API calls each frame. Additional (dynamic) objects are then rendered. In this case, you would not need to clear the back buffer or depth buffer.

Clearing of buffers and swapping the back buffer to the front buffer are all platform- and API-specific operations, so the virtual functions are pure and derived classes must implement them.

3.3.5 OBJECT DRAWING

The drawing of objects is supported by the following interface functions:

```
virtual bool BeginScene ();
virtual void EndScene ();
void DrawScene (VisibleSet& rkVisibleSet);
void Draw (Geometry* pkGeometry);
virtual void DrawElements () = 0;
void ApplyEffect (ShaderEffect* pkEffect, bool& rbPrimaryEffect);
```

The functions BeginScene and EndScene bound any block of code that contains drawing calls. Some graphics APIs might require predraw setup and postdraw cleanup. The main drawing routine in the sample applications is DrawScene. The input is the set of potentially visible objects. This set is generated by the scene graph culling system; see Section 4.5 for details. The function Draw is called indirectly by DrawScene for each geometric primitive of the scene, but Draw is a public function that can be used for drawing screen-space polygons; for example, GUI elements to be drawn on top of a rendered scene. The function ApplyEffect is called only by Draw. The object that is being drawn can have multiple special effects attached to it. The ApplyEffect function is called for each effect. It is also called when lights are active. The function DrawElements is implemented by each derived-class renderer. It is called once all the resources are loaded and enabled for the geometric primitive.

The functions `Draw` and `ApplyEffect` are discussed in greater detail in Section 3.4. They hide a lot of the drawing subsystem, including loading and enabling resources, constructing vertex and index buffers, validating shader programs, setting shader constants, and setting up the texture samplers.

3.3.6 TEXT AND 2D DRAWING

Displaying text on top of a rendered scene is quite common. The renderer API has a few basic functions to support this.

```
virtual int LoadFont (const char* acFace, int iSize, bool bBold,
    bool bItalic) = 0;
virtual void UnloadFont (int iFontID) = 0;
virtual bool SelectFont (int iFontID) = 0;
virtual void Draw (int iX, int iY, const ColorRGBA& rkColor,
    const char* acText) = 0;
virtual void Draw (const unsigned char* aucBuffer) = 0;
```

A minimal amount of font handling is provided. Classes `WglRenderer` and `Dx9Renderer` use Microsoft Windows API calls to select fonts. Class `Ag1Renderer` (Macintosh) has only a default font; no support has been added yet to select other fonts. Class `GlxRenderer` (Linux/Unix) uses bitmapped fonts. Interfacing with the windowing system is not the emphasis of this book.

The last function `Draw` allows you to write an already created color buffer to the back buffer. I use this only to support 2D graphics applications. The renderers write 2D primitives to a memory buffer, which is then passed to `Draw` and swapped to the front buffer. The samples contain a few 2D applications if you want to see the details. The 2D system has only a minimum amount of support for drawing primitives (points, line segments, circles, rectangles, text).

3.3.7 MISCELLANEOUS

The functions listed next support miscellaneous operations of interest.

```
virtual void SetColorMask (bool bAllowRed, bool bAllowGreen,
    bool bAllowBlue, bool bAllowAlpha);
virtual void GetColorMask (bool& rbAllowRed, bool& rbAllowGreen,
    bool& rbAllowBlue, bool& rbAllowAlpha);

virtual void SetDepthRange (float fZMin, float fZMax) = 0;

virtual void EnableUserClipPlane (int i, const Plane3f& rkPlane) = 0;
virtual void DisableUserClipPlane (int i) = 0;
```

```

virtual void SetPostWorldTransformation (const Matrix4f& rkMatrix);
virtual void RestorePostWorldTransformation ();

virtual void SetWorldTransformation ();
virtual void RestoreWorldTransformation ();

void SetProjector (Camera* pkProjector);
Camera* GetProjector ();

virtual const char* GetExtension () const = 0;
virtual char GetCommentCharacter () const = 0;

```

The functions `SetColorMask` and `GetColorMask` allow you to specify which color channels are to be updated by the pixel shader; see Section 3.1.11 for details.

The function `SetDepthRange` allows you to modify the default depth range $[0, 1]$ to be something different. This is useful in some special effects; for example, the `PlanarReflections` sample sets the depth range to be $[1, 1]$ so that the depth buffer values are set to the maximum depth when rendering the plane containing the mirror. This allows other objects to be rendered correctly (in depth) when they are drawn on top of (or in front of) the mirror.

The six frustum planes are used for clipping, but the user can specify additional clipping planes. The functions `EnableUserClipPlane` and `DisableUserClipPlane` allow you to specify clipping planes. The input plane must be in model coordinates. It is transformed internally to camera coordinates to support clipping in clip space. The `PlanarReflections` sample also manipulates user-defined clip planes.

The model-to-clip transformation is a composition applied to a model-space point,

$$\mathbf{X}_{\text{model}} \mathbf{H}_{\text{world}} \mathbf{H}_{\text{view}} \mathbf{H}_{\text{proj}}$$

Some special effects have a need to transform the world-space points before they are further processed in view space or in clip space. This is accomplished by inserting an addition transformation into the composition,

$$\mathbf{X}_{\text{model}} \mathbf{H}_{\text{world}} \mathbf{H}_{\text{postworld}} \mathbf{H}_{\text{view}} \mathbf{H}_{\text{proj}}$$

The effect provides $\mathbf{H}_{\text{postworld}}$ at the time it needs it. The functions `SetPostWorldTransformation` and `RestorePostWorldTransformation` are the hooks to allow you to specify such a transformation. The `PlanarReflections` sample uses this mechanism to insert a reflection matrix into the composition. The `PlanarShadows` sample uses the mechanism to insert a projection matrix into the composition. This matrix is the projection relative to the light source that generates the shadow.

The functions `SetWorldTransformation` and `RestoreWorldTransformation` are used internally by `Draw`. These set the world matrix stored in `Renderer`. The derived

classes also inform the graphics API about the new world matrix. The functions `SetProjector` and `GetProjector` are used by special effects that need to inform the renderer about a projector source, such as for projected textures, projected shadows, and shadow maps.

The functions `GetExtension` and `GetCommentCharacter` are implemented by the derived classes to provide information used for loading shader programs from disk and parsing them. All shader programs have an extension of `.wmisp` (Wild Magic Shader Program), but each graphics API has its own version of a program. The OpenGL versions have the compound extension `.ogl.wmisp`. The `GetExtension` function for this renderer returns `ogl`. The Direct3D renderer function returns `dx9` and the software renderer returns `sft`. The programs are parsed to obtain information about input variables, output variables, and shader constants, including which registers have been assigned to them. This information occurs in comments in the program files, each comment line starting with a specific character. This character is returned by a call to `GetCommentCharacter`. The OpenGL renderer returns the character `#`. The Direct3D and software renderers return the character `/`.

3.3.8 RESOURCE MANAGEMENT

The majority of `Renderer` is designed and built for resource management. Minimally, the derived-class renderers must specify the maximum number of various objects supported by the graphics hardware. These numbers are accessible by the interface functions

```
int GetMaxLights () const;
int GetMaxColors () const;
int GetMaxTCoords () const;
int GetMaxVShaderImages () const;
int GetMaxPShaderImages () const;
int GetMaxStencilIndices () const;
int GetMaxUserClipPlanes () const;
```

In the fixed-function pipeline, the number of lights was usually limited to eight. However, shader programming allows you to have as many lights as you want. That said, I implemented the renderers to limit the number of lights to eight, not that I want to limit you to a fixed number, but I cannot imagine a situation where you need so many dynamic lights affecting a single geometric primitive at one time.

The maximum number of colors is two for all the renderers. Vertex colors were typically stored as a single array of RGBA colors, but the evolution of graphics hardware led to APIs providing primary storage for diffuse colors and secondary storage for specular colors. From a practical perspective, I view these as providing two sets of vertex colors, so shader programs can be written to use the two sets. Processing of colors by the graphics APIs has potential pitfalls to be aware of. For example, the

Wild Magic colors are stored as `ColorRGB` or `ColorRGBA`, both having floating-point channels with values in $[0, 1]$. The OpenGL and software renderers want vertex data with color channels in $[0, 1]$. However, the Direct3D renderer wants the colors packed into a 32-bit quantity, 8 bits per channel. You must watch out for side effects that the graphics API generates when handling colors, such as internally clamping color values that are out of range. If you are using the vertex colors just for storage of noncolor quantities, it might be better to use texture coordinates for storage instead.

The function `GetMaxTCords` specifies how many sets of texture coordinates are allowed in the vertex and pixel programs. The function `GetMaxPShaderImages` specifies how many texture image units a pixel program supports. Although you might think these are the same, they need not be. For example, on an NVIDIA GeForce 6800 graphics card, the number of texture coordinate sets is eight but the number of texture image units is 16. If you use nine or more images, you must share some of the texture coordinate sets. The function `GetMaxVShaderImages` specifies how many texture image units a *vertex program* supports. Originally, vertex programs had no access to texture samplers—now they do.

The function `GetMaxStencilIndices` tells you how many distinct stencil values you can have. If you have an 8-bit stencil buffer, this number is 256. The function `GetMaxUserClipPlanes` returns the number of user-defined clip planes that you may enable. OpenGL reports six or more, but usually only six. This number is in addition to the six frustum planes, so in most cases the OpenGL renderer allows 12 clipping planes of which you can supply six.

Resource Loading and Releasing

Resources needed for rendering can exist in many places on a computer. The art content for scenes resides on disk (hard disk, CD-ROM, DVD, and so on). It is loaded into system memory, a relatively slow process. Eventually, some of this data must be sent to the graphics hardware to reside in video memory (VRAM). On many desktop computers, you have AGP memory. Transfers from AGP memory to VRAM are faster than transfers from system memory to VRAM. Generally, static geometry is cached in VRAM to avoid the bottleneck of constantly sending data across a bus to the graphics hardware. Dynamic geometry that changes infrequently can be stored in AGP memory; the idea is that you can modify the data as quickly as you can when it is in system memory, but the transfer time from AGP memory to VRAM is shorter than from system memory. The key to performance is to have your data in the right form, in the right place, and at the right time when it is needed for rendering.

The resource management system of `Renderer` is designed to help you with this. The relevant interface functions are

```
typedef void (Renderer::*ReleaseFunction)(Bindable*);
typedef void (Renderer::*ReleasePassFunction)(int, Bindable*);
```

```

void LoadAllResources (Spatial* pkScene);
void ReleaseAllResources (Spatial* pkScene);
void LoadResources (Geometry* pkGeometry);
void ReleaseResources (Geometry* pkGeometry);
void LoadResources (ShaderEffect* pkEffect);
void ReleaseResources (ShaderEffect* pkEffect);
void LoadVProgram (VertexProgram* pkVProgram);
void ReleaseVProgram (Bindable* pkVProgram);
void LoadPProgram (PixelProgram* pkPProgram);
void ReleasePProgram (Bindable* pkPProgram);
void LoadTexture (Texture* pkTexture);
void ReleaseTexture (Bindable* pkTexture);
void LoadVBuffer (int iPass, const Attributes& rkIAttr,
    VertexBuffer* pkVBuffer);
void ReleaseVBuffer (int iPass, Bindable* pkVBuffer);
void LoadIBuffer (IndexBuffer* pkIBuffer);
void ReleaseIBuffer (Bindable* pkIBuffer);

virtual void OnLoadVProgram (ResourceIdentifier& rpID,
    VertexProgram* pkVProgram) = 0;
virtual void OnReleaseVProgram (ResourceIdentifier* pkID) = 0;
virtual void OnLoadPProgram (ResourceIdentifier& rpID,
    PixelProgram* pkPProgram) = 0;
virtual void OnReleasePProgram (ResourceIdentifier* pkID) = 0;
virtual void OnLoadTexture (ResourceIdentifier& rpID,
    Texture* pkTexture) = 0;
virtual void OnReleaseTexture (ResourceIdentifier* pkID) = 0;
virtual void OnLoadVBuffer (ResourceIdentifier& rpID,
    const Attributes& rkIAttr, VertexBuffer* pkVBuffer) = 0;
virtual void OnReleaseVBuffer (ResourceIdentifier* pkID) = 0;
virtual void OnLoadIBuffer (ResourceIdentifier& rpID,
    IndexBuffer* pkIBuffer) = 0;
virtual void OnReleaseIBuffer (ResourceIdentifier* pkID) = 0;

```

The important resources are vertex buffers, index buffers, vertex programs, pixel programs, and textures. All of these are represented by classes shown here together with their base classes:

```

VertexBuffer : Object, Bindable
IndexBuffer  : Object, Bindable
VertexProgram : Program : Object, Bindable
PixelProgram  : Program : Object, Bindable
Texture       : Bindable

```

The graphics APIs allow you to load a resource into VRAM for later use. Each API provides you with a handle so that when you need the resource at a later time, you just let the API know the handle. The form of the handle depends on the API. I have hidden that by the class `Bindable`. This class also provides a two-way communication channel between the renderer and the resource. If the resource is released by the application, whether explicitly or because the resource is being destroyed, the renderer must be informed that the resource is going away so that the renderer can free up any internal data that was associated with the resource.

The `Load*` and `Release*` functions exist for each type of resource. The functions associated with vertex buffers have an input called `iPass`. This refers to the index of a pass within a multipass drawing operation. Each pass might have different vertex attribute requirements, so the format of a vertex buffer potentially varies per pass. This set of load and release functions is used internally by the `Draw` and `ApplyEffect` functions.

The load and release functions for the resources are called automatically by the drawing system; thus, a load occurs late in the drawing process. Once loaded, the resource is never loaded again unless in the meantime you have released it. In an application with a large amount of data to load, you most likely will see a slowdown of the frame rate. This is undesirable, so the interface also has higher-level load and release functions. These are designed to allow you to load and release at any time you like. For example, you might preload an entire level of a game while the user is watching a cut scene to keep him distracted. The interface allows you to load the resources associated with a single effect (shader programs and textures), with a geometry object (vertex and index buffers, resources associated with effects attached to the object), and with an entire scene (resources associated with every geometry object in the scene).

As an example of loading, consider the vertex program loader

```
void Renderer::LoadVProgram (VertexProgram* pkVProgram)
{
    ResourceIdentifier* pkID = pkVProgram->GetIdentifier(this);
    if (!pkID)
    {
        OnLoadVProgram(pkID,pkVProgram);
        pkVProgram->OnLoad(this,&Renderer::ReleaseVProgram,pkID);
    }
}
```

The first line of code involves a call to the `Bindable` function `GetIdentifier`. The input is the renderer since a resource can be bound to multiple renderers. The return value is a pointer to a resource identifier. The identifier is opaque—you should not care what it is, only that the pointer is null or nonnull. If the vertex program had been loaded earlier by the renderer, the returned pointer is nonnull and there is no work to be done. On the first load, though, the return value is null. In this

case, the vertex program must be loaded. Loading from disk to system memory is independent of platform in Wild Magic, but loading from system memory to VRAM (or AGP memory) is dependent on platform. Each derived class implements the virtual function `OnLoadVProgram` to load from system memory to VRAM. The final call is to the `Bindable` function `OnLoad`. The renderer (pointer) and identifier are passed to the function, but so is a function pointer. This function is called whenever the resource is about to be deleted, say, during a destructor call, to give the renderer a chance to free any associated data with the resource.

My intent is not to focus on the details within the graphics APIs, but just to give you an idea of how they compare, the implementations for `OnLoadVProgram` are listed here (without some error-handling details). The OpenGL implementation is

```
void OpenGLRenderer::OnLoadVProgram (ResourceIdentifier*& rpID,
    VertexProgram* pkVProgram)
{
    VProgramID* pkResource = WM4_NEW VProgramID;
    rpID = pkResource;
    const char* acProgramText =
        pkVProgram->GetProgramText().c_str();
    int iProgramLength = (int)strlen(acProgramText);
    glEnable(GL_VERTEX_PROGRAM_ARB);
    glGenProgramsARB(1,&pkResource->ID);
    glBindProgramARB(GL_VERTEX_PROGRAM_ARB,pkResource->ID);
    glProgramStringARB(GL_VERTEX_PROGRAM_ARB,
        GL_PROGRAM_FORMAT_ASCII_ARB,iLength,acProgram);
    glDisable(GL_VERTEX_PROGRAM_ARB);
}
```

The Direct3D implementation is

```
void Dx9Renderer::OnLoadVProgram (ResourceIdentifier*& rpID,
    VertexProgram* pkVProgram)
{
    VProgramID* pkResource = WM4_NEW VProgramID;
    rpID = pkResource;
    const char* acProgramText =
        pkVProgram->GetProgramText().c_str();
    int iProgramLength = (int)strlen(acProgramText);

    LPD3DXBUFFER pkCompiledShader = 0;
    LPD3DXBUFFER pkErrors = 0;
    D3DXAssembleShader(acProgramText,iProgramLength,0,0,0,
        &pkCompiledShader,&pkErrors);
}
```



```

m_pqDevice->CreateVertexShader(
    (DWORD*)(pkCompiledShader->GetBufferPointer()),
    &pkResource->ID);

if (pkCompiledShader)
{
    pkCompiledShader->Release();
}
if (pkErrors)
{
    pkErrors->Release();
}
}

```

The software renderer implementation is

```

void SoftRenderer::OnLoadVProgram (ResourceIdentifier*& rpkID,
    VertexProgram* pkVProgram)
{
    VProgramID* pkResource = WM4_NEW VProgramID;
    rpkID = pkResource;
    pkResource->OAttr = pkVProgram->GetOutputAttributes();
    stdext::hash_map<std::string,VProgram>::iterator pkIter =
        ms_pkVPrograms->find(pkVProgram->GetName());
    pkResource->ID = pkIter->second;
}

```

Releasing a resource is handled similarly.

```

void Renderer::ReleaseVProgram (Bindable* pkVProgram)
{
    ResourceIdentifier* pkID = pkVProgram->GetIdentifier(this);
    if (pkID)
    {
        OnReleaseVProgram(pkID);
        pkVProgram->OnRelease(this);
    }
}

```

If the return value from `GetIdentifier` is null, the resource is not currently loaded in the graphics system, so there is nothing to release. If the return value is nonnull, the resource must be released, the mechanism dependent on platform. Each derived class implements the virtual function `OnReleaseVProgram`. The final call is to the `Bindable`

function `OnRelease`, which just deletes the binding between the resource and the renderer.

The other load and release functions are similar except for the loading of vertex buffers. That function is

```
void Renderer::LoadVBuffer (int iPass,
    const Attributes& rkIAttr, VertexBuffer* pkVBuffer)
{
    // Search for a compatible vertex buffer that was used
    // during previous passes.
    ResourceIdentifier* pkID;
    for (int i = 0; i <= iPass; i++)
    {
        pkID = pkVBuffer->GetIdentifier(this,i);
        if (pkID)
        {
            if (rkIAttr.IsSubsetOf(*(Attributes*)pkID))
            {
                // Found a compatible vertex buffer in video
                // memory.
                return;
            }
        }
    }

    // The vertex buffer is encountered the first time.
    const Attributes& rkVBAAttr = pkVBuffer->GetAttributes();
    assert(rkIAttr.GetPChannels() == 3 &&
        rkVBAAttr.GetPChannels() == 3);
    if (rkIAttr.HasNormal())
    {
        assert(rkIAttr.GetNChannels() == 3 &&
            rkVBAAttr.GetNChannels() == 3);
    }

    OnLoadVBuffer(pkID,rkIAttr,pkVBuffer);
    pkVBuffer->OnLoad(this,iPass,&Renderer::ReleaseVBuffer,
        pkID);
}
```

A multipass effect might require different vertex attributes per pass. In this case, a single vertex buffer does not suffice. It is possible that two or more passes can share a single vertex buffer, so the first part of the load function iterates over the

bindings between the vertex buffer and the resources associated with the renderer. If a compatible resource is found, that one is used as the internal vertex buffer.

Resource Enabling and Disabling

Once a resource is loaded, it needs to be enabled for use by the graphics system. The enabling is done on each drawing pass. Once the object is drawn, the resources are disabled. The relevant interface functions are

```
void EnableVProgram (VertexProgram* pkVProgram);
void DisableVProgram (VertexProgram* pkVProgram);
void EnablePProgram (PixelProgram* pkPProgram);
void DisablePProgram (PixelProgram* pkPProgram);
void EnableTexture (Texture* pkTexture);
void DisableTexture (Texture* pkTexture);
ResourceIdentifier* EnableVBuffer (int iPass,
    const Attributes& rkIAAttr);
void DisableVBuffer (int iPass, ResourceIdentifier* pkID);
void EnableIBuffer ();
void DisableIBuffer ();

virtual void OnEnableVProgram (ResourceIdentifier* pkID) = 0;
virtual void OnDisableVProgram (ResourceIdentifier* pkID) = 0;
virtual void OnEnablePProgram (ResourceIdentifier* pkID) = 0;
virtual void OnDisablePProgram (ResourceIdentifier* pkID) = 0;
virtual void OnEnableTexture (ResourceIdentifier* pkID) = 0;
virtual void OnDisableTexture (ResourceIdentifier* pkID) = 0;
virtual void OnEnableVBuffer (ResourceIdentifier* pkID) = 0;
virtual void OnDisableVBuffer (ResourceIdentifier* pkID) = 0;
virtual void OnEnableIBuffer (ResourceIdentifier* pkID) = 0;
virtual void OnDisableIBuffer (ResourceIdentifier* pkID) = 0;
```

The first block of functions consists of wrappers that load or release the resources. For example, the function for enabling a texture has the simple form

```
void Renderer::EnableTexture (Texture* pkTexture)
{
    LoadTexture(pkTexture);
    ResourceIdentifier* pkID = pkTexture->GetIdentifier(this);
    OnEnableTexture(pkID);
}
```

The `On*` virtual functions are implemented by the derived-class renderers. For example, the function `OnEnableTexture` has the responsibility of informing the graphics

APIs about the texture state, such as filter mode, mipmap mode, wrap mode, border color, and so on. The function `OnDisableTexture` does nothing in the OpenGL and software renderers, but the Direct3D renderer disables the appropriate texture unit.

The enabling of shader programs involves an initial block of code, as shown for textures. This block lets the graphics API know the text string representation of the program. However, additional work must be performed to set the registers with the actual shader constants. The interface support for this subsystem is quite extensive.

```
enum // ConstantType
{
    CT_RENDERER,
    CT_NUMERICAL,
    CT_USER
};

virtual void SetVProgramConstant (int eCType,
    int iBaseRegister, int iRegisterQuantity,
    float* afData) = 0;

virtual void SetPProgramConstant (int eCType,
    int iBaseRegister, int iRegisterQuantity,
    float* afData) = 0;

enum { SC_QUANTITY = 38 };
typedef void (Renderer::*SetConstantFunction)(int,float*);
static SetConstantFunction ms_aoSCFunction[SC_QUANTITY];
void SetRendererConstant (RendererConstant::Type eRCType,
    float* afData);

// The operations are
// 0 = matrix
// 1 = transpose of matrix
// 2 = inverse of matrix
// 3 = inverse transpose of matrix
void GetTransform (Matrix4f& rkMat, int iOperation,
    float* afData);
void SetConstantWMatrix (int iOperation, float* afData);
void SetConstantVMatrix (int iOperation, float* afData);
void SetConstantPMatrix (int iOperation, float* afData);
void SetConstantWVMatrix (int iOperation, float* afData);
void SetConstantVPMatrix (int iOperation, float* afData);
void SetConstantWVPMatrix (int iOperation, float* afData);
```

```

// These functions do not use the option parameter, but the
// parameter is included to allow for a class-static array
// of function pointers to handle all shader constants.
void SetConstantMaterialEmissive (int, float* afData);
void SetConstantMaterialAmbient (int, float* afData);
void SetConstantMaterialDiffuse (int, float* afData);
void SetConstantMaterialSpecular (int, float* afData);
void SetConstantFogColor (int, float* afData);
void SetConstantFogParameters (int, float* afData);
void SetConstantCameraModelPosition (int, float* afData);
void SetConstantCameraModelDirection (int, float* afData);
void SetConstantCameraModelUp (int, float* afData);
void SetConstantCameraModelRight (int, float* afData);
void SetConstantCameraWorldPosition (int, float* afData);
void SetConstantCameraWorldDirection (int, float* afData);
void SetConstantCameraWorldUp (int, float* afData);
void SetConstantCameraWorldRight (int, float* afData);
void SetConstantProjectorModelPosition (int, float* afData);
void SetConstantProjectorModelDirection (int, float* afData);
void SetConstantProjectorModelUp (int, float* afData);
void SetConstantProjectorModelRight (int, float* afData);
void SetConstantProjectorWorldPosition (int, float* afData);
void SetConstantProjectorWorldDirection (int, float* afData);
void SetConstantProjectorWorldUp (int, float* afData);
void SetConstantProjectorWorldRight (int, float* afData);
void SetConstantProjectorMatrix (int, float* afData);

// These functions set the light state. The index iLight is
// between 0 and 7 (eight lights are currently supported).
void SetConstantLightModelPosition (int iLight, float* afData);
void SetConstantLightModelDirection (int iLight, float* afData);
void SetConstantLightWorldPosition (int iLight, float* afData);
void SetConstantLightWorldDirection (int iLight, float* afData);
void SetConstantLightAmbient (int iLight, float* afData);
void SetConstantLightDiffuse (int iLight, float* afData);
void SetConstantLightSpecular (int iLight, float* afData);
void SetConstantLightSpotCutoff (int iLight, float* afData);
void SetConstantLightAttenuation (int iLight, float* afData);

```

Shader constants come in three flavors:

1. *Renderer constants.* These include matrix transformations such as the world, view, and projection matrices. Any compositions are allowed, as well as transposes,

inverses, and inverse transposes. These also include light and material parameters, fog parameters, and camera and projector coordinate frame information. The 38 functions mentioned in the previous code block implement setting the registers for these constants. The current values of the renderer constants are automatically used.

2. *Numerical constants.* Direct3D requires you to set various numerical constants used by the shader programs. OpenGL and the software renderer do not.
3. *User-defined constants.* When writing shader programs, you can have constants that you will modify within your own application code. These tend to be related to the physical model of what you are trying to draw. For example, a refraction effect requires you to specify an index of refraction. This will be a shader constant that is used to generate the correct visual effect.

The vertex program enabler is

```
void Renderer::EnableVProgram (VertexProgram* pkVProgram)
{
    LoadVProgram(pkVProgram);
    ResourceIdentifier* pkID = pkVProgram->GetIdentifier(this);
    OnEnableVProgram(pkID);

    // Process the renderer constants.
    int i;
    for (i = 0; i < pkVProgram->GetRCQuantity(); i++)
    {
        RendererConstant* pkRC = pkVProgram->GetRC(i);
        SetRendererConstant(pkRC->GetType(),pkRC->GetData());
        SetVProgramConstant(CT_RENDERER,pkRC->GetBaseRegister(),
            pkRC->GetRegisterQuantity(),pkRC->GetData());
    }

    // Process the numerical constants.
    for (i = 0; i < pkVProgram->GetNCQuantity(); i++)
    {
        NumericalConstant* pkNC = pkVProgram->GetNC(i);
        SetVProgramConstant(CT_NUMERICAL,pkNC->GetRegister(),1,
            pkNC->GetData());
    }

    // Process the user-defined constants.
    for (i = 0; i < pkVProgram->GetUCQuantity(); i++)
    {
        UserConstant* pkUC = pkVProgram->GetUC(i);
```

```

        SetVProgramConstant(CT_USER, pkUC->GetBaseRegister(),
                           pkUC->GetRegisterQuantity(), pkUC->GetData());
    }
}

```

The program is loaded to VRAM (if not already there) and enabled for use. The remainder of the code loops over three arrays of constants, making calls to `SetVProgramConstant`. This function is pure virtual, so each derived class implements it. Each graphics API has functions that are called to actually set the registers associated with the constants.

The first loop sets the renderer constants and has more work to do than the other loops. It is necessary to determine which renderer constant needs to be set. When the vertex program is loaded from disk, it is parsed to obtain information about the renderer constants, including what type they are. The class that stores information about the constants to be used by the parser is `RendererConstant`. The interface is large but contains mainly enumerations for the various types (e.g., matrix, camera parameter, light parameter). The class also stores strings that name the renderer constants. These strings are used by the parser. As it turns out, naming conventions are necessary for renderer constants when writing Cg or HLSL shader programs. The static array

```
std::string RendererConstant::ms_kStringMap[];
```

stores the strings. If your shader program requires as input the matrix that maps model-space points to clip space, the corresponding shader constant must be named `WVPMatrix`. When the parser encounters this name, it will find a corresponding entry in `ms_kStringMap` and assign the associated enumeration to the `RendererConstant` object. This enumeration is passed to the `Renderer` function `SetRendererConstant` via `pkRC->GetType()`, and the matching `Renderer::SetConstant*` function is looked up and called.

Catalogs

When resources are loaded from disk to system memory to satisfy the needs of an object, it is possible that other objects share these resources. When one of these objects is encountered, it is an inefficient use of time to load the resource again. It is also an inefficient use of memory because you would have two copies of the resource in memory, which means you are not really sharing it.

To support sharing, the engine provides various *catalogs* of resources. The managed resources are vertex programs, pixel programs, and images. The related classes are `VertexProgramCatalog`, `PixelProgramCatalog`, and `ImageCatalog`, respectively.

All catalogs provide the ability to insert and remove items. Although you may do so explicitly, the engine will automatically handle the catalog management. For

example, if you create an Image object, you are required to provide a name for it. The Image object is automatically inserted into the image catalog. When the Image object is destroyed, the catalog is updated accordingly. Shader programs are handled in the same manner.

When a resource must be loaded from system memory to VRAM (or AGP memory), the catalog is searched first. If the resource is found in the catalog, it exists in system memory and is loaded to VRAM. If the resource is not found, an attempt is made to load it from disk to system memory. If this is successful, then the resource is loaded from system memory to VRAM. If the resource is not found on disk, a default resource is used. In the case of images, the default is an awful magenta color, something that hopefully will catch your attention during development and testing. The default vertex program only transforms the model-space position to clip space. The default pixel program creates a magenta color.

The idea of this system is to have a *memory hierarchy*. The levels are disk, system memory, and video memory. AGP memory is used at the discretion of the graphics drivers, so you tend not to have control over this level. For drawing purposes, if a resource is already in video memory, the renderer is good to go. If it is not in video memory but in system memory, a load to video memory must occur first. If it is not in system memory but on disk, a load from disk to system memory occurs, followed by a load from system memory to video memory. Finally, if it is not on disk, a default is used. Hopefully, this only occurs during development—a bug in your application that needs to be fixed before shipping the final product. The load and release functions described previously allow you to guarantee that resources are in video memory when needed, but you can also rely on loading to occur automatically on demand.

You can have multiple catalogs for each type of resource, but only one of these catalogs may be active at a single time. Each catalog class has a static data member that stores a pointer to the active catalog. This was a design choice just to get some type of catalog system working. It is possible to enhance the design by searching multiple catalogs rather than just the active one.

3.4 THE HEART OF THE RENDERER

The driving force behind the design of the rendering system is the need for a general-purpose and powerful drawing function for a geometric primitive. This function is `Renderer::Draw(Geometry*)`. It must support the following:

- Allow primitives with any topology such as points, polylines, triangle meshes.
- Allow for a primitive to override any currently active global render state.
- Set up the transformations in the geometric pipeline regardless of whether a perspective or orthographic projection is used.
- Support dynamic lighting when lights are present in the scene. These lights are in addition to any specifically used within a user-written shader program.

- Allow the geometric primitive to have one or more special effects. Each effect can implement a single-pass drawing operation or a multipass drawing operation.

Achieving these goals is the heart of the rendering system.

In version 3 and earlier of Wild Magic, geometric primitives were required to have a single effect attached to them, each effect implementing a single-pass drawing operation. Adding a multipass effect to the engine required you to create an *Effect* object *and* add a new pure virtual function to *Renderer* for that effect. Each derived-class renderer had to implement this virtual function. Examples that shipped with the engine included bump-mapping, spherical environment mapping, projected shadows, and planar reflections. The problem with this approach is that it is quite cumbersome to add multipass effects to the engine; that is, the engine is not easily extensible when it comes to adding new effects to it. Moreover, the rendering system can change frequently. It is desirable to have a core rendering system that supports multipass operations and multiple effects per primitive without having to modify the renderers.

Wild Magic version 4 implements such a rendering system. Prior to version 4, an *Effect*-derived class stored the function pointer to the *Renderer* function whose job it was to draw the geometric primitive with that effect. The roles are reversed now. If an *Effect*-derived class has special needs for multipass drawing, it implements a *Draw* function that is called by the renderer. This drawing function is called by the core rendering system and it calls *Renderer* functions as needed. This section provides a detailed description of the system from a top-down approach, which motivates why you would want each of the subsystems you encounter along the way.

3.4.1 DRAWING A SCENE

The top-level data structure for organizing objects to be drawn is a *scene graph*. For the purposes of this section, just think of the scene graph as a tree of nodes, where the leaf nodes represent the geometric primitives and the interior nodes represent a grouping of primitives based on spatial proximity. More details about scene graphs are found in Chapter 4. During program execution, scene graphs are passed to the culling system to produce potentially visible sets of objects. Each set is passed to the renderer for drawing. The top-level function for drawing is

```
void Renderer::DrawScene (VisibleSet& rkVisibleSet);
```

At its simplest level, the potentially visible set is a collection of geometric primitives whose order is determined by the depth-first traversal of the scene by the culler. *DrawScene* iterates over this set, calling *Renderer::Draw(Geometry*)* for each object.

Sophisticated effects, though, require a more complex system than just iterating over the geometric primitives. For example, projected planar shadows have the concept of a shadow caster (e.g., a biped character) and a plane to receive the shadow. Multiple passes must be made over the relevant portions of the scene. The shadow

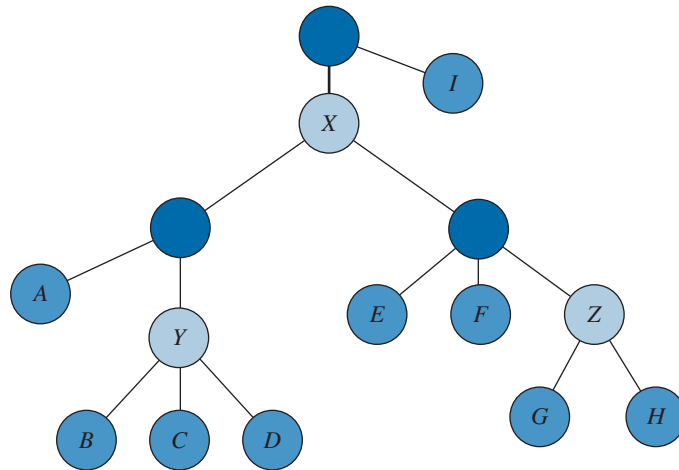


Figure 3.7 A scene graph with nine geometric primitives (leaf nodes) labeled A through I. Three interior nodes have global effects labeled X, Y, and Z. Primitives A through H are influenced by effect X. Primitives B, C, and D are influenced by effect Y. Primitives G and H are influenced by effect Z. Primitive I is not influenced by a global effect.

is generated from the perspective of a light projector, but the shadow caster must be drawn from the perspective of the camera. An effect that is assigned (attached) to a geometric primitive, which is represented as a leaf node of a scene, is thought of as a *local effect*. An effect such as projected shadows is assigned (attached) to a grouped set of primitives, which is represented as a subtree of a scene. The root node of the subtree is what the effect is attached to. Thus, the subtree is in the scope of the effect. This type of effect is thought of as a *global effect*.

The culler essentially flattens a scene graph into a list of objects to draw. When one or more global effects are present, the list must contain additional information to help control how the renderer draws the objects in the list. Figure 3.7 shows a scene graph with multiple global effects. The ensuing discussion explains how the global effects are stored in the list.

The potentially visible set computed by the culler is listed next as an array. A subscript of 0 on a global effect indicates the start of the scope of the effect. A subscript of 1 indicates the end of the scope. For example, objects B, C, and D are between Y_0 and Y_1 , so they are influenced by the global effect Y. They are also between X_0 and X_1 , so they are influenced as well by the global effect X. The set elements X_i , Y_i , and Z_i are *sentinels*, not drawable objects. The sentinels are used by the renderer to control the rendering of drawable objects.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Object	X_0	A	Y_0	B	C	D	Y_1	E	F	Z_0	G	H	Z_1	X_1	I

DrawScene implements the traversal of the potentially visible set and passes subsets of objects to the Draw functions of the global effects. The incremental steps are listed using pseudocode.

```

globalEffectStack = {};
startIndexStack = {};
globalEffectStack = {X};           // found  $X_0$ , push
startIndexStack = {1};             // push first index scope X
finalIndex = 1;                    // last index scope X
globalEffectStack = {Y,X};         // found  $Y_0$ , push
startIndexStack = {3,1};           // push first index scope Y
finalIndex = 3;                    // last index scope Y
finalIndex = 4;                    // last index scope Y
finalIndex = 5;                    // last index scope Y
globalEffectStack = {X};           // found  $Y_1$ , pop
startIndex = 3;                    // pop startIndexStack
startIndexStack = {1};             // ...
Y.Draw(startIndex,finalIndex);     // draw objects 3 to 5
finalIndex = 7;                    // last index scope X
finalIndex = 8;                    // last index scope X
globalEffectStack = {Z,X};         // found  $Z_0$ , push
startIndexStack = {10,1};          // push first index scope Z
finalIndex = 11;                   // last index scope Z
globalEffectStack = {X};           // found  $Z_1$ , pop
startIndex = 10;                   // pop startIndexStack
startIndexStack = {1};             // ...
Z.Draw(startIndex,finalIndex);     // draw objects 10 and 11
globalEffectStack = {};            // found  $X_1$ , pop
startIndex = 1;                    // pop startIndexStack
startIndexStack = {};              // ...
X.Draw(startIndex,finalIndex);     // draw objects 1 to 11
finalIndex = 14;                   // no global scope
Draw(finalIndex);                  // draw object 14 (local)

```

EXERCISE 3.8

Without looking at the Renderer source code, write pseudocode for the general process of handling global effects in a scene graph. ■

3.4.2 DRAWING A GEOMETRIC PRIMITIVE

The `Renderer::Draw(Geometry*)` function is listed next. I will explain it a piece at a time.

```
void Renderer::Draw (Geometry* pkGeometry)
{
    m_pkGeometry = pkGeometry;

    SetGlobalState(m_pkGeometry->States);
    SetWorldTransformation();

    EnableIBuffer();

    bool bPrimaryEffect = true;
    if (m_pkGeometry->LEffect)
    {
        ApplyEffect(m_pkGeometry->LEffect,bPrimaryEffect);
    }

    const int iEffectQuantity = m_pkGeometry->GetEffectQuantity();
    for (int iEffect = 0; iEffect < iEffectQuantity; iEffect++)
    {
        ShaderEffect* pkEffect =
            DynamicCast<ShaderEffect>(m_pkGeometry->GetEffect(iEffect));
        ApplyEffect(pkEffect,bPrimaryEffect);
    }

    DisableIBuffer();

    RestoreWorldTransformation();
    RestoreGlobalState(m_pkGeometry->States);

    m_pkGeometry = 0;
}
```

A member pointer, `m_pkGeometry`, is used to temporarily reference the geometric primitive that is being drawn. This is simply for convenience. The functions that set the matrix renderer constants need to access the transformations associated with the primitive. The enabling of vertex buffers also needs to access the primitive to get its vertex buffer.

The renderer maintains a set of the active global states (alpha blending, depth buffering, materials, and so on). The geometric primitive might have a need to override some of these. The call to `SetGlobalStates` allows the override. The call to

`RestoreGlobalState` restores the global state that was active before the drawing call was initiated.

The geometric primitive stores its model-space to world-space transformation. `SetWorldTransformation` passes this information to the graphics APIs so they may set their internal matrices. `RestoreWorldTransformation` restores the internal matrices to their values before the drawing call.

The topology of the geometric primitive (i.e., the index array) does not change during a drawing call and is independent of the number of drawing passes. The index buffer for the geometric primitive is loaded (if necessary) and enabled in the graphics APIs via the call to `EnableIBuffer`. After drawing, the index buffer is disabled by the call to `DisableIBuffer`.

If the geometric primitive is affected by lights in the scene, independent of the shader effects attached to the primitive, the lighting is applied first to the primitive. The data member `LEffect` is nonnull when dynamic lighting is needed. That effect is applied to the primitive via the call to `ApplyEffect`. When multiple effects must be applied to a primitive, the first effect is invariably applied so that the color buffer values are replaced by the effect colors. However, the effects occurring after the first one must be blended into the color buffer. The only way for an effect to know this is to pass it a Boolean flag, `bPrimaryEffect`, letting it know whether or not it is the first effect (primary effect).

The special effects directly attached to the geometric primitive are applied one at a time. These are necessarily `ShaderEffect` objects, which themselves directly use shader programs. Global effects are derived from the base class `Effect`. Each effect is applied to the primitive via the call to `ApplyEffect`. The Boolean flag `bPrimaryEffect` lets each effect know whether or not it is the first effect to be drawn.

3.4.3 APPLYING AN EFFECT

The `Renderer::ApplyEffect` encapsulates most of the resource management that was described earlier in this chapter. The source code follows. I will explain it a piece at a time.

```
void Renderer::ApplyEffect (ShaderEffect* pkEffect,
    bool& rbPrimaryEffect)
{
    const int iPassQuantity = pkEffect->GetPassQuantity();
    for (int iPass = 0; iPass < iPassQuantity; iPass++)
    {
        pkEffect->LoadPrograms(iPass,m_iMaxColors,
            m_iMaxTCords,m_iMaxVShaderImages,
            m_iMaxPShaderImages);

        pkEffect->SetGlobalState(iPass,this,rbPrimaryEffect);
    }
}
```

```

VertexProgram* pkVProgram =
    pkEffect->GetVProgram(iPass);
EnableVProgram(pkVProgram);

PixelProgram* pkPProgram =
    pkEffect->GetPProgram(iPass);
EnablePProgram(pkPProgram);

const int iPTQuantity =
    pkEffect->GetPTTextureQuantity(iPass);
int iTexture;
for (iTexture = 0; iTexture < iPTQuantity; iTexture++)
{
    EnableTexture(
        pkEffect->GetPTTexture(iPass,iTexture));
}

const Attributes& rkIAttr =
    pkVProgram->GetInputAttributes();
ResourceIdentifier* pkID = EnableVBuffer(iPass,rkIAttr);

DrawElements();

DisableVBuffer(iPass,pkID);

for (iTexture = 0; iTexture < iPTQuantity; iTexture++)
{
    DisableTexture(
        pkEffect->GetPTTexture(iPass,iTexture));
}

DisablePProgram(pkPProgram);

DisableVProgram(pkVProgram);

pkEffect->RestoreGlobalState(iPass,this,
    rbPrimaryEffect);
}

rbPrimaryEffect = false;
}

```

The function is designed to support multipass operations. The number of passes is read into `iPassQuantity`, which is 1 for a single-pass effect. Let's take a look at what happens for each pass.

The first statement in the loop has the effect of loading its shader programs for the specified pass. The programs must be loaded first because (1) the vertex buffer for the pass is enabled based on the program inputs and (2) the global-state setting that occurs in the second statement might need to access the samplers related to the program. The resource limits are passed to the `LoadPrograms` function because it is written to be independent of renderer type and platform. The shader programs about to be loaded might require more resources than the active renderer can provide. In this case, some action is required on the effect's part to deal with the problem. My choice for now is just to use the default shader programs. A commercial engine should provide some fallback mechanism to shaders that produce similar results, probably of lesser quality, but ones that can be obtained with the renderer's limited resources.

After the programs are loaded, the effect is given a chance to set global render state. The most common operation is to set alpha blending functions, especially when the effect is not the primary one. Notice that the Boolean flag `bPrimaryEffect` is passed to the effect's `SetGlobalState` function so that it can decide whether or not it needs to enable alpha blending. The global state must be set before enabling programs because the programs set sampler state for samplers about to be enabled.

Next, the vertex and pixel programs are enabled (and loaded to VRAM the first time if necessary). Any textures associated with the effect are enabled. These can be specific to each pass. Similarly, the vertex buffer associated with the geometric primitive for this specific pass is enabled.

Once all the resources are loaded and enabled, we are ready to tell the graphics APIs to draw the geometric primitive. This occurs in the `DrawElements` call. The OpenGL and Direct3D implementations for `DrawElements` are very short, but they hide a lot of details. These details were discussed in terms of software rendering in Section 3.1.

After drawing, all the resources are disabled, global state is restored to its previous state, and the effect processing is complete.

3.4.4 LOADING AND PARSING SHADER PROGRAMS

The entry point to loading and parsing shader programs for an effect is the function `ShaderEffect::LoadPrograms` that is called in the `Renderer::ApplyEffect` function. The first part of the source code is

```
void ShaderEffect::LoadPrograms (int iPass, int iMaxColors,
                                int iMaxTCoords, int iMaxVShaderImages,
                                int iMaxPShaderImages)
{
    Program* pkVProgram = m_kVShader[iPass]->GetProgram();
    Program* pkPProgram = m_kPShader[iPass]->GetProgram();
    assert((pkVProgram != 0) == (pkPProgram != 0));
}
```

```

    if (pkVProgram)
    {
        // The programs have already been loaded.
        return;
    }

    VertexProgramPtr spkVProgram =
        VertexProgramCatalog::GetActive()->Find(
            m_kVShader[iPass]->GetShaderName());

    PixelProgramPtr spkPProgram =
        PixelProgramCatalog::GetActive()->Find(
            m_kPShader[iPass]->GetShaderName());

    <remainder of implementation>
}

```

The function checks to see if the shader programs have already been loaded. In the common case they should be, since loading should occur only once, because of either the first attempt to draw or an explicit resource load forced by the application code.

When the programs have not been loaded, the function checks the catalogs for the vertex and pixel programs. For example, the Find function for vertex programs is

```

VertexProgram* VertexProgramCatalog::Find (
    const std::string& rkProgramName)
{
    if (rkProgramName == ms_kNullString
        || rkProgramName == ms_kDefaultString)
    {
        return StaticCast<VertexProgram>(
            m_spkDefaultVProgram);
    }

    // Attempt to find the program in the catalog.
    stdext::hash_map<std::string,VertexProgram*>::iterator
        pkIter = m_kEntry.find(rkProgramName);
    if (pkIter != m_kEntry.end())
    {
        // The program exists in the catalog, so return it.
        return pkIter->second;
    }

    // Attempt to load the program from disk.
    assert(m_cCommentChar);
}

```



```

VertexProgram* pkProgram = VertexProgram::Load(
    rkProgramName,m_kRendererType,m_cCommentChar);
if (pkProgram)
{
    // The program exists on disk. Insert it into the
    // catalog.
    m_kEntry.insert(
        std::make_pair(rkProgramName,pkProgram));
    return pkProgram;
}

// The program does not exist. Use the default program.
return StaticCast<VertexProgram>(m_spkDefaultVProgram);
}

```

The catalog maintains a hash map of pairs of string names and vertex programs in order to guarantee fast lookups. As the comments indicate, if the string name specifies a program already in the map, that program had been loaded earlier and the pointer to it is returned by the Find function. If it does not exist in the map, it must be loaded from disk. An attempt is made to load it. If it is loaded from disk, the program is inserted into the hash map so it may be shared later by other effects. If it cannot be found on disk, the default vertex program is returned.

The catalogs are designed to allow any type renderer to be used so that they may store programs for OpenGL, Direct3D, or the software renderer. The platform-specific information is stored in the data members `m_kRendererType`, which is one of `ogl`, `dx9`, or `sft`. The data member `m_cCommentChar` is either `#` for OpenGL or `/` for Direct3D and the software renderer. The application layer provides default catalogs so that you do not have to know the catalog system even exists. The catalogs are created premain, but the renderer has not yet been created. Assignment of the catalog data members is deferred until the renderer is created in the function `WindowApplication::OnInitialize`. The catalog classes provide the function `SetInformation` to support the deferred assignment. The inputs to this function are obtained via calls to `Renderer::GetExtension` for `m_kRendererType` and to `Renderer::GetCommentCharacter` for `m_cCommentChar`.

The function source code for the vertex program load is

```

VertexProgram* VertexProgram::Load (
    const std::string& rkProgramName,
    const std::string& rkRendererType,
    char cCommentPrefix)
{
    std::string kFilename = std::string("v_") +
        rkProgramName + std::string(".") + rkRendererType +
        std::string(".wmisp");
}

```

```

VertexProgram* pkProgram = WM4_NEW VertexProgram;
bool bLoaded = Program::Load(kFilename,cCommentPrefix,
    pkProgram);
if (!bLoaded)
{
    WM4_DELETE pkProgram;
    return 0;
}

pkProgram->SetName(rkProgramName.c_str());
VertexProgramCatalog::GetActive()->Insert(pkProgram);
return pkProgram;
}

```

The first line of code creates the full name for the disk file that stores the vertex program. All such programs have the prefix `v_`. The pixel programs have the prefix `p_`. If you create your own shader programs, you need to follow this convention for file names.

A vertex program is created using the default constructor.² The static member function `Program::Load` fills in the data members of the vertex program by actually loading the file and parsing it. Assuming a successful load, the vertex program is inserted into the catalog for access by other effects sharing that program, and the vertex program is returned to the calling function.

That brings us to the workhorse of the loading system, the function `Program::Load`. The class `Program` is designed solely to support loading a shader program from disk and parsing it to identify its inputs, outputs, shader constants and associated registers, and texture samplers. The public interface for the class is

```

class Program : public Object, public Bindable
{
    // Abstract base class.
    virtual ~Program ();

    // Member read-only access.
    const std::string& GetProgramText () const;
    const Attributes& GetInputAttributes () const;
    const Attributes& GetOutputAttributes () const;

```

2. If you look at the source code for `VertexProgram` and `Program`, you will notice that the default constructors are protected. From an application's perspective, the only way to create a shader program is by loading from disk. Eventually, I will add the subsystem for procedurally generated shader programs and for shader stitching.

```

// Access to renderer constants.
int GetRCQuantity () const;
RendererConstant* GetRC (int i);
RendererConstant* GetRC (RendererConstant::Type eType);

// Access to numerical constants.
int GetNCQuantity () const;
NumericalConstant* GetNC (int i);

// Access to user constants.
int GetUCQuantity () const;
UserConstant* GetUC (int i);
UserConstant* GetUC (const std::string& rkName);

// Access to samplers.
int GetSIQuantity () const;
SamplerInformation* GetSI (int i);
SamplerInformation* GetSI (const std::string& rkName);
}

```

The class is abstract, but designed to support only two derived classes: `VertexProgram` and `PixelProgram`. Once the protected `Load` function is called and the parsing successful, you have access to the program text string, the input attributes, and the output attributes. As mentioned previously, there are three types of shader constants: renderer constants, numerical constants (Direct3D only), and user-defined constants. These are all accessible through the public interface. Finally, you can access the number of samplers required by the program (possibly none) and the information for each sampler.

The parsing code in `Program::Load` is quite lengthy, so I will not go into details here. I wrote the code manually, inferring the grammar rules for the assembly text files by inspection. The parser is not complete. Currently, it does not support shader programs whose inputs are arrays. The manual creation of a parser for shader programs is probably not the best approach in a production environment. Better to use tools such as Lex, Yacc, Flex, and Bison (e.g., <http://dinosaur.compilertools.net>) to automatically generate the parser code from a grammar. That way you can focus on maintaining the grammar rules as shader programs evolve or to support many types of shader programs instead of just Cg or HLSL.

EXERCISE 3.9

Modify the parsing code to support shader program inputs that are arrays of values. ■

Attributes

When you look at sample shader programs, regardless of which language they are written in, you see that you must pass in various quantities such as positions, normals, colors, and texture coordinates. Wild Magic encapsulates the concept of attributes in a class called, well, `Attributes`. This turns out to be a very convenient class. It is used to store input and output attributes for shader programs, but it is also used for specifying the structure of a vertex buffer that is attached to a geometric primitive. The class also allows for comparing two sets of attributes to see if they are equal or at least compatible (in the sense of subset inclusion).

The `Attributes` class allows you to set the number of channels for positions, normals, colors, and texture coordinates. Currently, you may have positions and normals with three or four channels. The three-channel values are the usual 3D quantities you have in your models, (x, y, z) . The four-channel values are to support homogeneous points and vectors, (x, y, z, w) . These are set via

```
// 3 or 4 channels, (xyz,xyzw)
void Attributes::SetPChannels (int iPChannels);
void Attributes::SetNChannels (int iNChannels);
```

You are allowed an arbitrary number of color attributes, even though the shader programs currently support up to two colors. You specify each *unit* and the number of channels the unit supports, which is one through four.

```
// 1 to 4 channels, (r,rg,rgb,rgba)
void Attributes::SetCChannels (int iUnit, int iCChannels);
```

Similarly, you are allowed an arbitrary number of texture coordinate attributes. You specify each unit and the number of channels the unit supports, which is one through four.

```
// 1 to 4 channels, (s,st,str,strq)
void Attributes::SetTChannels (int iUnit, int iTChannels);
```

The attributes are organized internally to have the ordering

```
position, normal, color[0], color[1],..., tcoord[0], tcoord[1],...
```

The organization occurs automatically, so you need not worry about calling the `Attributes` functions in any particular order.

Renderer Constants

The renderer constants were discussed previously. The class `RendererConstant` has a set of enumerations that name the renderer constants that a shader program will use. The class also has a corresponding set of string names for these constants. The parser in `Program::Load` compares the names of shader constants found in the shader program file to the string names of `RendererConstant`. When it finds a match, the constant is added to an array of `RendererConstant` objects in the `Program` object. In addition to storing the value of the constant, the parser also discovers which register (or registers) has been assigned to the constant and stores that information. The shader constants and registers are accessed by the renderers when enabling a shader program and setting the registers it needs with the constant values.

Numerical Constants

The class `NumericalConstant` represents a numerical value that is needed in a shader program. Only the `Direct3D` renderer requires you to specify explicitly the numerical constants. `OpenGL` and the software renderer automatically process numerical constants. The parser in `Program::Load` creates `NumericalConstant` objects, even for `OpenGL` and the software renderer, and stores them in an array that the `Program` object manages. The stored information includes both the constant values and the registers assigned to them. The numerical constants are accessed by the renderers when enabling a shader program.

User-Defined Constants

The class `UserConstant` represents a user-defined constant that is needed in a shader program. `Renderer` constants and numerical constants are automatically assigned to registers when a shader program is enabled; the application code has no responsibility to make this happen. However, user-defined constants are quantities that the application wants to modify as needed. Thus, the `UserConstant` class has a richer interface than `RendererConstant` or `NumericalConstant` to allow communication between the application and the internal storage of the user-defined constants.

The constructor for the user-defined constant requires you to supply a string name for the constant. The string name allows an application program to query a shader program to obtain a handle to the user-defined constant. The application may then modify that constant as needed. The question, though, is, Where is the constant stored?

The `Shader` class provides storage for the user-defined constants. This is a natural thing to do for the following reasons. A `Program` object may be shared by multiple shaders. What needs to be shared is the *program text string*, which is the assembly source code for the program. Just like a C function or a C++ function, the shader

function is “called” by the graphics system for the currently active geometric primitive. The inputs to the function depend on the geometric primitive. Moreover, those inputs include shader constants that can vary per call to the shader program. The Program function cannot store the constants—they must be passed to the program by a client of that program, in our case a Shader object.

Multiple instances of a Shader class may be created by the application and attached to different geometric primitives. The instances share the Program object, but each instance can have its own set of user-defined constants. Thus, the Shader class stores the Program object and an array of floating-point values for storage associated with the user-defined constants. The class also stores arrays of Texture objects and associated Image objects. The reason for this is the same as that for user-defined constants. A shader program that has texture sampler inputs will use whatever textures and images are currently enabled. The textures and images vary per object, even though the shader program does not.

Sampler Information

The texture samplers in a shader program have various information that needs to be identified by the parser in `Program::Load`. First, each sampler in a program has a name associated with it so you can use the sampler within the program. Second, it is necessary to know what type of sampler is required. The types include samplers for 1D, 2D, and 3D textures. A couple of types support some common effects. A *cube sampler* supports cube environment maps. Essentially, this is a collection of six 2D samplers to handle the six faces of the cube map. A *projector sampler* supports projected textures and shadow maps. Essentially, this is a 2D sampler, but the texture coordinates are in homogeneous points. Third, the compiled shader program includes the texture/sampler unit that will do the actual sampling when the program is called. The unit number must be remembered so that the graphics engine can set up the unit accordingly when the textures are enabled during a drawing call.

The class that stores all this information is `SamplerInformation`. The Program class maintains an array of these objects that are associated with the program itself.

Associating Data with the Program

Once `Program::Load` has loaded and parsed the program, the Program object has the following:

- A program text string containing the assembly instructions to be executed by the graphics system.
- A set of inputs stored as an `Attributes` object.
- A set of outputs stored as an `Attributes` object.

- An array of `RendererConstant` objects, which always has at least one element—a transformation that maps the input positions to clip space since the rasterizer needs this information. Each object has a type identifier, data storage, and the registers to be loaded with the data.
- An array of `NumericalConstant` objects, which the Direct3D renderer needs (but which OpenGL and the software renderer handle automatically).
- An array of `UserConstant` objects. Each object has a name for ease of access by an application, a pointer to the storage location for the data, and the registers to be loaded with the data.
- An array of `SamplerInformation` objects. Each object has a name for the sampler, the type of the sampler, and the texture unit number that refers to the actual unit that will do the sampling.

Each `Program` object is managed by a `Shader` object; specifically, a `VertexProgram` object is managed by a `VertexShader` object and a `PixelProgram` object is managed by a `PixelShader` object. The `Shader` object has the following:

- A string name (mainly for lookup in the program catalogs).
- A `Program` object.
- Storage for user-defined constants that the program uses.
- An array of image names. The images themselves are actually stored in an image catalog. The names are used to look up the images in the catalog.
- An array of `Texture` objects. These are the textures associated with the samplers of the program.

A `ShaderEffect` object manages arrays of `VertexShader` and `PixelShader` objects. Each pair of vertex and pixel shaders corresponds to a drawing pass. The `ShaderEffect` object also manages an array of `AlphaState` objects, each specifying how a pass is blended into the rendered results of previous passes.

The question now is, How do the user-defined shader constants, the textures, and the images become associated with the shader program? This brings us full circle to where we began—loading a shader effect during a drawing operation. Let's summarize what we have so far. A geometric primitive is to be drawn. The global state, transformations, and index buffer have been set up. Now we apply a `ShaderEffect` to the primitive via the function `ApplyEffect`. For each pass, the following steps are taken.

- The `ShaderEffect` object has a `VertexShader` object and a `PixelShader` object. The `VertexShader` object has a `VertexProgram` object, and the `PixelShader` object has a `PixelProgram` object. An attempt is made to load the programs via the function `ShaderEffect::Load`.

- The load function fetches the programs from the program catalogs, which either already have them stored in memory or must load them from disk.
- When the programs are loaded from disk, they are parsed to identify the input and output attributes, the renderer constants, the numerical constants, the user-defined constants, and the texture samplers.

Then we associate actual attributes, constants, textures, and images with the programs. And finally, the last part of `ShaderEffect::Load` is

```
void ShaderEffect::LoadPrograms (int iPass, int iMaxColors,
                                int iMaxTCoords, int iMaxVShaderImages,
                                int iMaxPShaderImages)
{
    <first portion of implementation>

    m_kVShader[iPass]->OnLoadProgram(spKVProgram);
    m_kPShader[iPass]->OnLoadProgram(spKPProgram);
    OnLoadPrograms (iPass, spKVProgram, spKPProgram);
}
```

After the programs are loaded, each is given a chance to perform postload operations via the `OnLoadProgram` function. This function associates the data with the program. The source code is

```
void Shader::OnLoadProgram (Program* pkProgram)
{
    assert(!m_spkProgram && pkProgram);
    m_spkProgram = pkProgram;

    // The data sources must be set for the user constants.
    // Determine how many float channels are needed for the
    // storage.
    int iUCQuantity = m_spkProgram->GetUCQuantity();
    int i, iChannels;
    UserConstant* pkUC;
    for (i = 0, iChannels = 0; i < iUCQuantity; i++)
    {
        pkUC = m_spkProgram->GetUC(i);
        assert(pkUC);
        iChannels += 4*pkUC->GetRegisterQuantity();
    }
    m_kUserData.resize(iChannels);
```



```

// Set the data sources for the user constants.
for (i = 0, iChannels = 0; i < iUCQuantity; i++)
{
    pkUC = m_spkProgram->GetUC(i);
    assert(pkUC);
    pkUC->SetDataSource(&m_kUserData[iChannels]);
    iChannels += 4*pkUC->GetRegisterQuantity();
}

// Load the images into the textures. If the image is
// already in system memory (in the image catalog), it
// is ready to be used. If it is not in system memory,
// an attempt is made to load it from disk storage. If
// the image file does not exist on disk, a default
// magenta image is used.
int iSIQuantity = m_spkProgram->GetSIQuantity();
m_kImageNames.resize(iSIQuantity);
m_kTextures.resize(iSIQuantity);
for (i = 0; i < iSIQuantity; i++)
{
    Image* pkImage = ImageCatalog::GetActive()->Find(
        m_kImageNames[i]);
    assert(pkImage);
    m_kTextures[i].SetImage(pkImage);
    m_kTextures[i].SetSamplerInformation(
        m_spkProgram->GetSI(i));
}
}

```

The first part of the code iterates over the UserConstant objects and determines exactly how many floating-point channels are needed to store the data for the user-defined constants. The storage in Shader is resized to handle this amount.

The second part of the code sets the pointers in the UserConstant objects to point to the correct locations in the array where their data is stored. When the application needs to access this data, it must make the appropriate query using the constant's string name to obtain a pointer to the data's storage location.

The third part of the code resizes the image and texture arrays to be large enough to handle what the program requires. The texture images are loaded by querying the active image catalog. This catalog behaves similarly to the program catalogs. If the image is already in memory, a pointer is returned to that image. If the image is not in memory, an attempt is made to load it from disk. If the attempt is successful, the image is placed in a hash map to allow sharing, and the image pointer is returned. If the image is not found on disk, the default image is returned (a magenta color). Each

Texture object is given its texture image and corresponding sampler information. The image and sampler information are accessed by the graphics system when the texture is loaded/enabled for drawing.

There is also a companion function to `OnLoadProgram` named `OnReleaseProgram`. This function simply decrements its reference to the shader program. If the shader object is the last object to have a reference to the program, the program is automatically destroyed.

The last line of code in `ShaderEffect::Load` is a call to a virtual function `ShaderEffect::OnLoadPrograms`. This gives objects from a `ShaderEffect`-derived class to perform postload operations. The most common operation is to provide local storage for user-defined constants. For example, if you look at the sample application file,

`GeometricTools\WildMagic4\SampleGraphics\Iridescence\IridescenceEffect.cpp`

you will see that the `IridescenceEffect` class has implemented this virtual function:

```
void IridescenceEffect::OnLoadPrograms (int, Program* pkVProgram, Program*)
{
    pkVProgram->GetUC("InterpolateFactor")->SetDataSource(m_afInterpolate);
}
```

The first input parameter is the drawing pass index. This class implements a single-pass effect, so the pass index is always zero. It is not needed here, so according to the ANSI standards, the formal parameter name is omitted. The second input parameter is the vertex program, which we access in this program. The last input parameter is the pixel program, but we do not access it here, so the formal parameter is omitted.

The `IridescenceEffect` class stores a user-defined constant named `m_afInterpolate`. The constant is one of the inputs to the Cg shader program, namely, the input `InterpolateFactor`:

```
void IridescenceVProgram
(
    in float4      kModelPosition : POSITION,
    in float3      kModelNormal : NORMAL,
    in float2      kInBaseTCoord : TEXCOORD0,
    out float4     kClipPosition : POSITION,
    out float2     kOutBaseTCoord : TEXCOORD0,
    out float      fOutInterpolateFactor : TEXCOORD1,
    out float3     kWorldNormal : TEXCOORD2,
    out float3     kEyeDirection : TEXCOORD3,
    uniform float4x4 WVPMatrix,
```

```

    uniform float4x4 WMatrix,
    uniform float3   CameraWorldPosition,
    uniform float    InterpolateFactor)
{
    // ... code goes here ...
}

```

The parser in `Program::Load` stored the name `InterpolateFactor` in a `UserConstant` object. The `ShaderEffect::OnLoadProgram` function provided storage for this constant and set a pointer in the `UserConstant` object to point to this storage. However, the function `IridescenceEffect::OnLoadPrograms` overrides this. It queries for the `UserConstant` object with the name `InterpolateFactor`, and reassigns the data pointer to point to the local copy `m_afInterpolate`. Now every time your application code needs the user-defined constant to change value, it does so by modifying `m_afInterpolate` through the member accessor `IridescenceEffect::SetInterpolate`. This is more efficient than having `SetInterpolate` query for the user-defined constant, access its data pointer, and assign the new value; that is, the query is an additional cost you want to avoid.

If the query is inefficient, why provide storage for user-defined constants in `Shader` in the first place? The `IridescenceEffect` class is created at development time, so it is easy enough to specify the user-defined constants within the class. However, for procedurally generated shader programs and for generic setup of effects through `ShaderEffect` (no relevant derived classes for you to manipulate), you need to have some storage automatically provided. Classes derived from `ShaderEffect` may be thought of as part of a fixed-function pipeline, but it is actually possible for Wild Magic to process all shader programming through `ShaderEffect` without any derived classes.

3.4.5 VALIDATION OF SHADER PROGRAMS

In the previous discussion, I talked about the first and last parts of `ShaderEffect::Load`. The first part loads the shader programs and the last part associates user-defined constants, textures, and images with the programs. I omitted discussion about the middle part of the program. Its job is to perform *validation*. There are three things you need to validate:

1. The geometric primitive's vertex buffer data must contain the input vertex attributes required by the vertex program.
2. The vertex program's output attributes must match the pixel program's input attributes.
3. The resources required by the vertex and pixel programs do not exceed the limits imposed by the renderer.

The first validation occurs in `Renderer::LoadVBuffer`, a function I mentioned in Section 3.3 without raising the issue of validation. That function has the block of code

```
ResourceIdentifier* pkID;
for (int i = 0; i <= iPass; i++)
{
    pkID = pkVBuffer->GetIdentifier(this,i);
    if (pkID)
    {
        if (rkIAttr.IsSubsetOf(*(Attributes*)pkID))
        {
            // Found a compatible vertex buffer in video memory.
            return;
        }
    }
}
```

The parameter `rkIAttr` is the `Attributes` object corresponding to the input attributes for the vertex program. The resource identifier is safely typecast to an `Attributes` object. This object corresponds to the attributes for the vertex buffer associated with the geometric primitive. The function `Attributes::IsSubsetOf` tests to see if the caller, `rkIAttr`, has attributes that are contained by the input to the function. That is, a check is made to see if the primitive's vertex buffer has at least the attributes required by the vertex program. If it does, then the vertex buffer may be used to load vertex data to video memory. Later, in `LoadVBuffer` there was a line of code

```
OnLoadVBuffer(pkID,rkIAttr,pkVBuffer);
```

The derived-class renderers have implementations for this function. They make a call to

```
int iChannels;
float* afCompatible;
pkVBuffer->BuildCompatibleArray(rkIAttr,false,iChannels,afCompatible);
```

At the time this is called, we know that the program input attributes, `rkIAttr`, are a subset of the vertex buffer's attributes. The vertex buffer is instructed to call `Attributes::BuildCompatibleArray`, which allocates an array `afCompatible` and fills it with data according to the attributes in `rkIAttr`. The false input has to do with how color data is packaged: `Direct3D` wants 8-bit color channels packed into a 32-bit quantity. `OpenGL` and the software renderer store the floating-point channels (in `[0, 1]`) directly. The packaged array is then handed to the graphics APIs to be uploaded to VRAM, returning to you a resource identifier that is used for later lookups of this vertex buffer.

The second and third items in the list of validations are performed by `ShaderEffect::Load`. The second item is implemented by the code block

```
std::string kDefault("Default");
const Attributes& rkV0Attr =
    spkVProgram->GetOutputAttributes();
const Attributes& rkPIAttr =
    spkPProgram->GetInputAttributes();
if (!rkV0Attr.Matches(rkPIAttr,false,true,true,true))
{
    // The output attributes of the vertex program and the
    // input attributes of the pixel program are incompatible.
    // Use the default shader objects.
    if (spkVProgram->GetName() != kDefault)
    {
        m_kVShader[iPass] = WM4_NEW VertexShader(kDefault);
        spkVProgram =
            VertexProgramCatalog::GetActive()->Find(kDefault);
    }

    if (spkPProgram->GetName() != kDefault)
    {
        m_kPShader[iPass] = WM4_NEW PixelShader(kDefault);
        spkPProgram =
            PixelProgramCatalog::GetActive()->Find(kDefault);
    }
}
```

The function `Attributes::Matches` compares the attributes of the vertex program output attributes, `rkV0Attr`, and the pixel program input attributes, `rkPIAttr`. The four Boolean inputs to this function indicate whether you should compare positions, normals, colors, or textures. Notice that the first Boolean input is `false`. The vertex program outputs clip-space coordinates for the vertex position to be used by the rasterizer. The pixel program has no need for this information, so the `Matches` function is instructed to ignore positions. However, the pixel program does potentially care about the normals, colors, and texture coordinates.

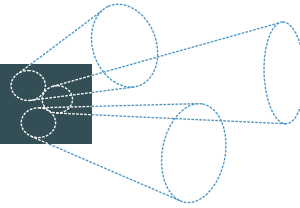
Mismatches between the geometric primitive's vertex buffer and the vertex program inputs may be handled by the graphics engine, but since the shader system of the graphics APIs does not allow intervention in the rasterization and feeding of the pixel programs, it is not possible programmatically to deal with mismatches between vertex program outputs and pixel program inputs. It is possible that the graphics drivers will deal with this. Regardless, my implementation handles a mismatch by ignoring the loaded shader programs, instead attaching the default shader programs.

Once you have a pair of compatible vertex and pixel programs, the final validation step checks to make sure that enough resources exist. The code block that handles this is

```
const Attributes& rkVIAttr = spkVProgram->GetInputAttributes();
if (rkVIAttr.GetMaxColors() > iMaxColors
|| rkVIAttr.GetMaxTCords() > iMaxTCords
|| rkVOAttr.GetMaxColors() > iMaxColors
|| rkVOAttr.GetMaxTCords() > iMaxTCords
|| rkPIAttr.GetMaxColors() > iMaxColors
|| rkPIAttr.GetMaxTCords() > iMaxTCords
|| spkVProgram->GetSIQuantity() > iMaxVShaderImages
|| spkPProgram->GetSIQuantity() > iMaxPShaderImages)
{
    // The renderer cannot support the requested resources.
    if (spkVProgram->GetName() != kDefault)
    {
        m_kVShader[iPass] = WM4_NEW VertexShader(kDefault);
        spkVProgram =
            VertexProgramCatalog::GetActive()->Find(kDefault);
    }

    if (spkPProgram->GetName() != kDefault)
    {
        m_kPShader[iPass] = WM4_NEW PixelShader(kDefault);
        spkPProgram =
            PixelProgramCatalog::GetActive()->Find(kDefault);
    }
}
```

This is why the active renderer's resource limits are passed to the `ShaderEffect::Load` function. The program inputs, outputs, and sampler information is compared to what the renderer can handle. If any resource exceeds the limits, the loaded shader programs are ignored and the default shader programs are used.



SCENE GRAPHS

Chapters 2 and 3 were about drawing geometric primitives to a window. This chapter is about organizing the primitives to efficiently feed the renderer and covers the two competing schools of thought; namely, that the organization should be spatial, where objects are grouped together based on their spatial proximity, or state based, where objects are grouped together based on their common render state. I will briefly discuss both concepts here, concluding that the data structure for object organization—called a *scene graph*—should support both aspects of spatial proximity and render-state coherency.

4.1 SCENE GRAPH DESIGN ISSUES

Most applications have a large number of objects. The simplest algorithm for drawing them is

```
scene = <set of objects in application>;  
for each object in scene do  
{  
    renderer.Draw(object);  
}
```

This is a very inefficient approach. Some of the objects are visible, but many are not. For the nonvisible objects, the renderer spends a lot of time determining that all their triangles are outside the view frustum. The majority of time is spent on culling back-facing triangles and clip-culling the front-facing triangles.

Assuming you have the ability to test whether an object is partially or totally inside the view frustum, a somewhat more efficient algorithm for drawing the objects is

```
scene = <set of objects in application>;
for each object in scene do
{
    if (frustum.ContainsSomeOf(object))
    {
        renderer.Draw(object);
    }
}
```

The premise is that the cost of function `ContainsSomeOf(object)` is much less than the cost of `renderer.Draw(object)` for nonvisible objects. Even this algorithm is potentially inefficient for a couple of reasons. There is no attempt to exploit either spatial coherency or render-state coherency.

- *Spatial coherency.* If the view frustum is much smaller than the world in which the objects live, you might have prior knowledge of where most of the objects are located. It is quite possible to partition the world into small cells. If you know the current cell (or cells) that contains the view frustum, you need only iterate over the content of that cell (or cells).

```
scene = <set of objects in application>;
partition = <set of cells partitioning the world>;
frustumCells = <set of cells overlapping the frustum>;
for each cell in frustumCells do
{
    for each object in cell do
    {
        if (frustum.ContainsSomeOf(object))
        {
            renderer.Draw(object);
        }
    }
}
if frustum moves then
{
    update frustumCells;
}
```

Such an organization helps you reduce the number of visibility tests for objects, but you have to design and create the organizational data structures.

- *Render-state coherency.* Changing render state during a drawing pass can be expensive, depending on what state you are changing. A simple iteration over potentially visible objects does not take this into account. If the objects were sorted according to their render state, you could improve the efficiency by

```
scene = <set of objects in application>;
sortedScene = <scene objects sorted by render state>;
for each object in sortedScene do
{
    if (frustum.ContainsSomeOf(object))
    {
        renderer.Draw(object);
    }
}
```

Now you have the responsibility for maintaining a data structure that keeps the objects sorted by render state (during add and remove operations).

Ideally, we should choose a data structure that supports both types of coherency. The problem is that the goals are at odds with each other. A collection of objects with the same render state might very well be scattered across the world. Similarly, a collection of objects close to each other might be spatially organized in a manner that requires constant switching of render state. There is a middle ground.

Organization by spatial proximity (geometric state) is the most natural choice for grouping objects, especially since we rely on artists to build the objects using 3D modeling packages. The classic data structure for grouping is a tree. The leaf nodes represent the geometric primitives and the interior nodes represent grouping of related objects. Objects such as articulated biped characters inherently require a tree-based data structure. The motion of the hand of a character is intimately tied into the motion of the wrist, elbow, and shoulder. Moreover, the motion is affected by the biped as a whole, whether moving through space (translation) or changing orientation (rotation). Organization by render state is not something intuitive to many artists. You could supply some guidelines on how to build objects to take advantage of render-state coherency, but I imagine this would be difficult to enforce. In practice, attempts to incorporate render-state coherency are usually postprocessing steps applied to the 3D models after the artists have done their job.

The fact that spatial coherency is more important in designing a data structure for objects in a scene is also supported by the driving force behind real-time rendering: The fastest way to draw a scene is to avoid drawing anything at all. If an object is not in the view frustum, the renderer should not be given that object to draw. Sorting everything by render state is irrelevant if you cannot quickly decide what is *not* visible.

I have heard the debates about which is better, organization by spatial coherency or by render-state coherency. The correct choice is to use whichever leads to better performance of your application. My experiences have led me to design a scene graph

to be centered around a spatial hierarchy, each node of the hierarchy introducing a coordinate system for the subtree whose root is that node. Each node maintains transformations and bounding volumes, the latter used for rapid culling by the view frustum. Minimizing the render-state changes is, of course, also desirable. My choice is to build a set of potentially visible objects first, followed by a preprocessing step to sort that set based on render state. This is a reasonable design as long as various subsystems of the scene graph management are built to be as modular as possible. Wild Magic's scene graph management contains four such subsystems:

1. *Updating geometric state.* Updating the spatial information in the hierarchy when geometric state changes, including vertex data changes, transformation changes, and scene graph topology changes.
2. *Updating render state.* Updating the information supplied in the hierarchy when the render state changes. This includes attaching and detaching of global states and scene graph topology changes.
3. *Culling pass.* Determining the set of potentially visible objects in a scene after geometric state changes.
4. *Drawing pass.* Drawing the set of potentially visible objects.

Wild Magic versions 1 and 2 already incorporated the updating geometric state and render state. However, these versions had the culling pass and drawing pass combined into a single pass. As a scene graph was traversed, nonvisible objects were culled (i.e., not drawn), but each potentially visible object was drawn as it was encountered. Such a design makes it nearly impossible to sort the objects by render state. Wild Magic version 3 fixed this problem to some extent by deferring the drawing. As each potentially visible object was encountered during a drawing pass, it was placed in a set of objects to be handed to the renderer for drawing once the entire scene was traversed. The renderer then had the option of sorting these objects by render state (or by any other criteria) before drawing them. Two problems with this design are that the renderer encapsulates the sorting, making it difficult to tap into the system without constantly modifying the renderer source code, and that the semantics of handling global effects (attached to interior nodes of the tree) are difficult to manage.

Wild Magic version 4, which is what ships with this book, includes a rewritten rendering system. In addition to switching from a fixed-function pipeline to a shader-based system, the culling pass and drawing pass have been separated. This allows for a producer-consumer model, where the potentially visible sets can be generated by one process (the producer) without regard to how the renderer (the consumer) draws the data or even when the renderer needs the data. This is a good separation of concerns, especially with threading in an application. The culling process can run in a thread separate from the drawing process. This is particularly useful for special effects such as shadow mapping, where two potentially visible sets are generated for a single drawing pass. The first set is relative to the camera; the second set is relative to the light source for deciding where a shadow is cast.

Another major benefit in separating the culling and drawing is that an intermediate step between the two passes can involve sorting based on render state. This is yet another modular subsystem that is orthogonal to the culling and drawing passes. And as mentioned in Section 3.4.1, the generation of potentially visible sets by the culling process includes sentinels for delimiting the scope of a global effect. A sorting process applied to a potentially visible set can pay attention to the sentinels, sorting objects only between a matching pair of sentinels.

4.1.1 THE CORE CLASSES

The classes in the scene graph management system that support a spatial tree (or hierarchy) of objects are `Spatial`, `Node`, and `Geometry`. The objects are organized by spatial relationships. Using a modeling package, an artist will create geometric data, usually in the form of points, polylines, and triangle meshes, and assign various visual attributes, including textures, lighting and materials, and shader effects. Additional information is also created by the artist. For example, keyframe data may be added to a biped structure for the purpose of animation of the character. Complicated models such as a biped are typically implemented by the model package using a scene graph hierarchy itself. For illustration, though, consider a simple, inanimate object such as a model of a wooden table.

Geometry

The table consists of geometric information in the form of a collection of model vertices. For convenience, suppose they are stored in an array $V[i]$ for $0 \leq i < n$. Most likely the table is modeled as a triangle mesh. The triangles are defined as triples of vertices, ordered in a consistent manner that allows you to say which side of the triangle is outward facing, from a display perspective, and which side is inward facing. A classical choice for outward-facing triangles is to use counterclockwise ordering: If an observer is viewing the plane of the triangle and that plane has a normal vector pointing to the side of the plane on which the observer is located, the triangle vertices are seen in a counterclockwise order in that plane. The triangle information is usually stored as a collection of triples of indices into the vertex array. Thus, a triple (i_0, i_1, i_2) refers to a triangle whose vertices are $(V[i_0], V[i_1], V[i_2])$. If dynamic lighting of the table is desired, an artist might additionally create vertex model normals, although in many cases it is sufficient to generate the normals procedurally. Other vertex attributes are added to the model as needed to support shader effects. Finally, the model units are possibly of a different size than the units used in the game's world, or the model is intended to be drawn in a different size than what the modeling package does. A model scale may be applied by the artist to accommodate these. This does allow for nonuniform scaling, where each spatial dimension may be scaled independently of the others. The region of space that the model occupies is represented by a model bound, typically a sphere or box or some other simple object, that

encloses all the vertices. This information can always be generated procedurally and does not require the artist's input. The model bound is useful for identifying whether or not the model is currently visible to an observer. All the model information created by the artist, or procedurally generated from what the artist produces, is encapsulated by the class `Geometry`.

Vertex and Index Buffers

In Wild Magic version 3, the vertex positions and normals were stored as separate arrays in the `Geometry` objects. The vertex colors and texture coordinates were stored in `Effect` objects to be used in various special effects, whether of the flavor of the fixed-function pipeline or a shader program. The separation of vertex attributes is not natural for an extensible shader-based system. In Wild Magic version 4, the vertex positions and any attributes for a `Geometry` object are stored in a *vertex buffer*. This is an “array of structs,” each structure containing all the information for a single vertex, as compared to the previous version that used a “struct of arrays.” The class representing this is called `VertexBuffer`. In previous versions of Wild Magic, the index array associated with the geometric primitive (points, polyline, triangle mesh) was also stored in the `Geometry` object, but now it is stored in what is called an *index buffer* represented by class `IndexBuffer`. Both buffer classes act as containers for data, but they also have an interface `Bindable` for associating the buffers (in system memory) with their counterparts used for drawing (in video memory).

Lights and Effects

In the new design, the `Geometry` object contains all the information needed by the renderer to correctly draw it. How the data is used depends on the special effects attached to the object or on global effects attached to predecessor nodes of the hierarchy. These effects include dynamic lighting, implied by `Light` objects attached to predecessor nodes in the hierarchy path leading to the `Geometry` object. The effects also include *local effects*, obtained by attaching `ShaderEffect` objects to the `Geometry` objects. Wild Magic version 3 and earlier only allowed one such effect per object. Wild Magic version 4 allows multiple effects. The drawing system is designed to automatically provide multipass rendering to handle these. `Geometry` objects are also affected by *global effects*, obtained by attaching `Effect` objects to predecessor nodes of the `Geometry` objects.

Spatial

Suppose that the artist was responsible for creating both a table and a room in which the table is placed. The table and room will most likely be created in separate modeling sessions. When working with the room model, it would be convenient to

load the already-created table model and place it in the room. The technical problem is that the table and room were created in their own, independent *coordinate systems*. To place the table, it must be translated, oriented, and possibly scaled. The resulting *local transformation* is a necessary feature of the final scene for the game. I use the adjective *local* to indicate that the transformation is applied to the table relative to the coordinate system of the room. That is, the table is *located* in the room, and the relationship between the room and table may be thought of as a *parent-child* relationship. You start with the room (the parent) and place the table (the child) in the room using the coordinate system of the room. The room itself may be situated relative to another object, for example, a house requiring a local transformation of the room into the coordinate system of the house. Assuming the coordinate system of the house is used for the game's world coordinate system, there is an implied *world transformation* from each object's model space to the world space. It is intuitive that the model bound for an object in model space has a counterpart in world space, a *world bound*, which is obtained by applying the world transformation to the model bound. The local and world transformations and the world bound are encapsulated by the class `Spatial`.

Node

The example of a house, room, and table has another issue that is partially related to the local and world transformations. The objects are ordered in a natural hierarchy. To make the example more illustrative, consider a house with two rooms, with a table and chair in one room, and a plate, fork, and knife placed on the table. The hierarchy for the objects is shown in Figure 4.1. Each object is represented by a node in the hierarchy.

The objects are all created separately. The hierarchy represents parent-child relationships regarding how a child object is placed relative to its parent object. The Plate, Knife, and Fork are assigned local transformations relative to the Table. The Table and

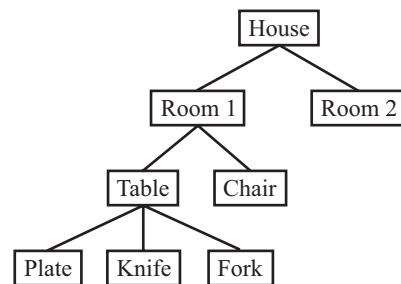


Figure 4.1 A hierarchy to represent a collection of related objects.

Chair are assigned local transformations relative to Room 1. Room 1 and Room 2 are assigned local transformations relative to the House. Each object has world transformations to place it directly in the world. If L_{object} is the local transformation that places the object in the coordinate system of its parent and W_{object} is the world transformation of the object, the hierarchy implies the following matrix compositions. The order of application to vectors (the vertices) is from right to left according to the conventions used in Wild Magic:

$$\begin{aligned}
 W_{\text{House}} &= L_{\text{House}} \\
 W_{\text{Room1}} &= W_{\text{House}} L_{\text{Room1}} = L_{\text{House}} L_{\text{Room1}} \\
 W_{\text{Room2}} &= W_{\text{House}} L_{\text{Room2}} = L_{\text{House}} L_{\text{Room2}} \\
 W_{\text{Table}} &= W_{\text{Room1}} L_{\text{Table}} = L_{\text{House}} L_{\text{Room1}} L_{\text{Table}} \\
 W_{\text{Chair}} &= W_{\text{Room1}} L_{\text{Chair}} = L_{\text{House}} L_{\text{Room1}} L_{\text{Chair}} \\
 W_{\text{Plate}} &= W_{\text{Table}} L_{\text{Plate}} = L_{\text{House}} L_{\text{Room1}} L_{\text{Table}} L_{\text{Plate}} \\
 W_{\text{Knife}} &= W_{\text{Table}} L_{\text{Knife}} = L_{\text{House}} L_{\text{Room1}} L_{\text{Table}} L_{\text{Knife}} \\
 W_{\text{Fork}} &= W_{\text{Table}} L_{\text{Fork}} = L_{\text{House}} L_{\text{Room1}} L_{\text{Table}} L_{\text{Fork}}
 \end{aligned}$$

The first equation says that the house is placed in the world directly. The local and world transformations are the same. The second equation says that Room 1 is transformed first into the coordinate system of the House and then is transformed to the world by the House's world transformation. The other equations have similar interpretations. The last one says that the Fork is transformed into the coordinate system of the Table, then transformed to the coordinate system of Room 1, then transformed to the coordinate system of the House, then transformed to the world coordinates. A path through the tree of parent-child relationships has a corresponding sequence of local transformations that are composited. Although each local transformation may be applied one at a time, it is more efficient to use the world transformation of the parent (already calculated by the parent) and the local transformation of the child to perform a single matrix product that is the world transformation of the child.

The grouping together of objects in a hierarchy is the role of the Node class. The compositing of transformations is accomplished through a depth-first traversal of the parent-child tree. Each parent node provides its world transformation to its child nodes in order for the children to compute their world transformations, naturally a recursive process. The transformations are propagated *down the hierarchy* (from root node to leaf nodes).

Each geometric object has a model bound associated with it. A node does not have a model bound per se, given that it only groups together objects, but it can be assigned a world bound. The world bound indicates that portion of space containing the collection of objects represented by the node. Keep in mind that the bound is a coarse measurement of occupation and that not all of the space contained in the bound is occupied by the object. A natural choice for the world bound of a node is any bound that contains the world bounds of its children. However, it is not necessary

that the world bound contain the child bounds. All that matters is that the *objects* represented by the child nodes are contained in the world bound. Once a world bound is assigned to a node, it is possible to define a model bound—the one obtained by applying the inverse world transformation to the world bound. A model bound for a node is rarely used, so the `Node` class does not have a data member to store this information. If needed, it can be computed on the fly from other data.

Each time local transformations are modified at a node in the scene, the world transformations must be recalculated by a traversal of the subtree rooted at that node. But a change in world transformations also implies a change in the world bounds. After the transformations are propagated down the hierarchy, new world bounds must be recomputed at the child nodes and propagated *up the hierarchy* (from leaf nodes to root node) to parent nodes so that they may also recompute their world bounds. The model bound is transformed to the world bound by the world transformation. The world transformation at a child node depends on its parent's world transformation. The composition of the transformations occurs during the downward pass through the hierarchy. The parent's world bound depends on the child's world bound. The recalculation of the world bounds occurs during the upward pass through the hierarchy. The downward and upward passes together are referred to as a *geometric update*, whose implementation details will be discussed later.

Controllers and Modifiers

The word *animation* tends to be used in the context of motion of characters or objects. I use the word in a more general sense to refer to any time-varying quantity in the scene. The engine has support for animation through *controllers*; the abstract base class is `Controller`.

The most common controllers are transform controllers—for example, keyframe controllers or inverse kinematic controllers. For keyframe controllers, an artist provides a set of positions and orientations for objects (i.e., for the nodes in the hierarchy that represent the objects). A keyframe controller interpolates the keyframes to provide smooth motion over time. For inverse kinematic controllers, the positions and orientations for objects are determined by constraints that require the object to be in certain configurations. For example, a hand on a character must be translated and rotated to pick up a glass. The controller selects the translations and rotations for the hand according to where the glass is.

Vertex and normal controllers are used for morphing and mesh deformation. Render-state controllers are used for animating just about any effect you like. For example, a controller could be used to vary the color of a light source. A texture may be animated by varying the texture coordinates associated with the texture and the object to which the texture applies. This type of effect is useful for giving the effect that a water surface is in motion.

Index controllers are less common, but are used to dynamically change the topology of a triangle mesh or strip. For example, continuous-level-of-detail algorithms may be implemented using controllers.

I use the term *modifier* to indicate additional semantics applied to a collection of vertices, normals, and indices. The Geometry class is a container for these items but is itself an abstract class. The main modifier is class TriMesh, which is derived from Geometry, and this class is used to provide indices to the base class. A similar example is class TriStrip, where the indices are implicitly created by the class and provided to the Geometry base class. In both cases, the derived classes may be viewed as index modifiers of the Geometry base class.

Other geometry-based classes may also be viewed as modifiers of Geometry, including points (class Polypoint) and polylines (class Polyline). Both classes may be viewed as vertex modifiers. Particle systems (base class Particles) are derived from class TriMesh. The particles are drawn as rectangular billboards (the triangle mesh stores the rectangles as pairs of triangles) and so may be thought of as index modifiers. However, the physical aspects of particles are tied into only the point locations. In this sense, particle systems are vertex modifiers of the Geometry class.

How one adds the concept of modifiers to an engine is up for debate. The controller system allows you to attach a list of controllers to an object. Each controller manages the animation of some member (or members) of the object. As you add new Controller-derived classes, the basic controller system need not change. This is a good thing since you may extend the behavior of the engine without having to rearchitect the core. Preserving old behavior when adding new features is related to the object-oriented principle called the *open-closed principle*. After building a system that, over time, is demonstrated to function as designed and is robust, you want it to be *closed* to further changes in order to protect its integrity. Yet you also want the system to be *open* to extension with new features. Having a core system such as the controllers that allows you to create new features and support them in the (closed) core is one way in which you can have both open and closed qualities.

The classical manner in which you obtain the open-closed principle, though, is through class derivation. The base class represents the closed portion of the system, whereas a derived class represents the open portion. Regarding modifiers, I decided to use class derivation to define the semantics. Such semantics can be arbitrarily complex—something not easily fitted by a system that allows a list of modifiers to be attached to an object. A derived class allows you to implement whatever interface is necessary to support the modifications. Controllers, on the other hand, have simple semantics. Each represents management of the animation of one or more object members, and each implements an update function that is called by the core system. The controller, list-based system is natural for such simple objects.

4.1.2 SPATIAL HIERARCHY DESIGN

The main design goal for class Spatial is to represent a coordinate system in space. Naturally, the class members should include the local and world transformations and the world bounding volume, as discussed previously. The Geometry and Node classes

themselves involve transformations and bounding volumes, so it is natural to derive these from `Spatial`. What is not immediately clear is the choice for having both classes `Spatial` and `Node`. In Figure 4.1, the objects `Table`, `Plate`, `Knife`, `Fork`, and `Chair` are `Geometry` objects. They all are built from model data; they all occupy a portion of space; and they are all transformable. The objects `House`, `Room 1`, and `Room 2` are grouping nodes. We could easily make all these `Spatial` objects, but not `Geometry` objects. In this scenario, the `Spatial` class must contain information to represent the hierarchy of objects. Specifically, each object must have a link to its parent object (if any) and links to its child objects. The links shown in Figure 4.1 represent both the parent and child links.

The concepts of grouping and of representing geometric data are effectively disjoint. If `Spatial` objects were allowed child objects, then by derivation so would `Geometry` objects. Thus, `Geometry` objects would have double duty, as representations of geometric data and as nodes for grouping related objects. The interface for a `Geometry` class that supports grouping as well as geometric queries will be quite complicated, making it difficult to understand all the behavior that objects from the class can exhibit. I prefer instead a *separation of concerns* regarding these matters. The interfaces associated with `Geometry` and its derived classes should address only the semantics related to geometric objects, their visual appearances, and physical properties. The grouping responsibilities are delegated instead to a separate class, in this case the class `Node`. The interfaces associated with `Node` and its derived classes address only the semantics related to the subtrees associated with the nodes. By separating the responsibilities, it is easier for the engine designer and architect to maintain and extend the separate types of objects (geometry types or node types).

My choice for separation of concerns leads to class `Spatial` storing the parent link in the hierarchy and to class `Node` storing the child links in the hierarchy. Class `Node` derives from `Spatial`, so in fact the `Node` objects have both parent and child links. Class `Geometry` also derives from `Spatial`, but geometry objects can occur only as leaf nodes in the hierarchy. This is the main consequence of the separation of concerns. The price you pay for having the separation and a clean division of responsibilities is that the hierarchy as shown in Figure 4.1 is not realizable in this scheme. Instead, the hierarchy may be structured as shown in Figure 4.2.

Two grouping nodes were added. The `Table Group` node was added because the `Table` is a geometric object and cannot be an interior node of the tree. The utensils (`Plate`, `Knife`, `Fork`) were children of the `Table`. To preserve this structure, the `Utensil Group` node was added to group the utensils together. To maintain the transformation structure of the original hierarchy, the `Table Group` is assigned the transformations the `Table` had, the `Table` is assigned the identity transformation, and the `Utensil Group` is assigned the identity transformation. This guarantees that the `Utensil Group` is in the same coordinate system that the `Table` is in. Consequently, the utensils may be positioned and oriented using the same transformations that were used in the hierarchy of Figure 4.1.

Alternatively, you can avoid the `Utensil Group` node and just make the utensils siblings of the `Table`. If you do this, the coordinate system of the utensils is now that of

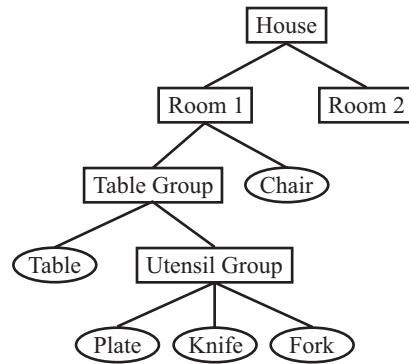


Figure 4.2 The new hierarchy corresponding to the one in Figure 4.1 when geometric objects can be only leaf nodes. Ellipses are used to denote geometric objects. Rectangles are used to denote grouping nodes.

the Table Group. The transformations of the utensils must be changed to ones relative to the coordinate system of the Table Group.

The portion of the interface for class `Spatial` relevant to the scene hierarchy connections is

```

class Spatial : public Object
{
public:
    virtual ~Spatial ();
    Spatial* GetParent ();

protected:
    Spatial ();
    Spatial* m_pkParent;

// internal use
public:
    void SetParent (Spatial* pkParent);
};
  
```

The default constructor is protected, making the class an abstract base class. The default constructor is implemented to support the streaming system. The class is derived from the root class `Object`, as are nearly all the classes in the engine. All of the root services are therefore available to `Spatial`, including run-time type information, sharing, streaming, and so on.

The parent pointer is protected, but read access is provided by the public interface function `GetParent`. Write access of the parent pointer is provided by the public interface function `SetParent`. That block of code is listed at the end of the class. My intention on the organization is that the public interface intended for the application writers is listed first in the class declaration. The public interface at the end of the class is tagged with the comment “internal use.” The issue is that `SetParent` is called by the `Node` class when a `Spatial` object is attached as the child of a node. No other class (or application) should call `SetParent`. If the method were put in the protected section to prevent unintended use, then `Node` cannot call the function. To circumvent this problem, `Node` can be made a friend of `Spatial`, thus allowing it access to `SetParent`, but disallowing anyone else to access it. In some circumstances, a `Node`-derived class might also need access to a protected member of `Spatial`. In the C++ language, friendship is not inherited, so making `Node` a friend of `Spatial` will not make the `Node`-derived class a friend of `Spatial`. To avoid the somewhat frequent addition of friend declarations to classes to allow restricted access to protected members, I decided to use the system of placing the restricted access members in public scope but tagging that block with the “internal use” comment to let programmers know that they should *not* use those functions.

The portion of the interface for class `Node` relevant to the scene hierarchy connections is

```
class Node : public Spatial
{
public:
    Node ();
    virtual ~Node ();

    int GetQuantity () const;
    int AttachChild (Spatial* pkChild);
    int DetachChild (Spatial* pkChild);
    SpatialPtr DetachChildAt (int i);
    SpatialPtr SetChild (int i, Spatial* pkChild);
    SpatialPtr GetChild (int i);

protected:
    std::vector<SpatialPtr> m_kChild;
};
```

The links to the child nodes are stored as an array of `Spatial` smart pointers. For a discussion of smart pointers and reference counting, see Section 18.5. Clearly, the pointers cannot be `Node` pointers, because the leaf nodes of the hierarchy are `Spatial`-derived objects (such as `Geometry`), but not `Node`-derived objects. The nonnull child pointers do not have to be contiguous in the array, so where the children are placed is up to the programmer.

The `AttachChild` function searches the pointer array for the first available empty slot and stores the child pointer in it. If no such slot exists, the child pointer is stored at the end of the array, dynamically resizing the array if necessary. This is an important feature to remember. For whatever reason, if you detach a child from a slot internal to the array and you do not want the next child to be stored in that slot, you must use the `SetChild` function because it lets you specify the exact location for the new child. The return value of `AttachChild` is the index into the array where the attached child is stored. The return value of `SetChild` is the child that was in the *i*th slot of the array before the new child was stored there. If you choose not to hang onto the return value, it is a smart pointer, in which case the reference count on the object is decremented. If the reference count goes to zero, the child is automatically destroyed.

Function `DetachChild` lets you specify the child, by pointer, to be detached. The return value is the index of the slot that stored the child. The vacated slot has its pointer set to `NULL`. Function `DetachChildAt` lets you specify the child, by index, to be detached. The return value is that child. As with `SetChild`, if you choose not to hang onto the return value, the reference count on the object is decremented and, if zero, the object is destroyed.

Function `GetChild` simply returns a smart pointer to the current child in the specified slot. This function is what you use when you iterate over an array of children and process them in some manner—typically something that occurs during a recursive traversal of a scene graph.

4.1.3 SHARING OF OBJECTS

The spatial hierarchy system is a *tree structure*; that is, each tree node has a single parent, except for a root node that has no parent. You may think of the spatial hierarchy as the skeleton for the scene graph. A scene graph really is an abstract graph because the object system supports sharing. If an object is shared by two other objects, effectively, there are two *instances* of the first object. The act of sharing the objects is called *instancing*. I do not allow instancing of nodes in a spatial hierarchy, and this is enforced by allowing a `Spatial` object to have only a single parent link. Multiple parents are not possible.¹ One of the questions I am occasionally asked is why I made this choice.

For the sake of argument, suppose that a hierarchy node is allowed to have multiple parents. A simple example is shown in Figure 4.3. The scene graph represents a house with two rooms. The rooms share the same geometric model data. The two rooms may be thought of as instances of the same model data. The implied structure is a directed acyclic graph (DAG). The `House` node has two directed arcs to the `Room` nodes. Each `Room` node has a directed arc to the `Room Contents` leaf node. The `Room Contents` are therefore shared. Reasons to share include reducing memory

1. *Predecessors* might be a better term to use here, but I will use the term *parents* and note that the links are directed from parent to child.

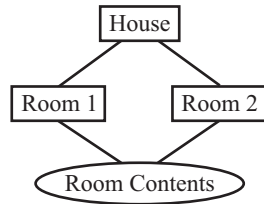


Figure 4.3 A scene graph corresponding to a house and two rooms. The rooms share the same geometric model data, called Room Contents.

usage for the game application and reducing your artist's workload in having to create distinct models for everything you can imagine in the world. The hope is that the user is not terribly distracted by the repetition of like objects as he navigates through the game world.

What are some of the implications of Figure 4.3? The motivation for a spatial hierarchy is to allow for positioning and orienting of objects via local transformations. The locality is important so that generation of content can be done independently of the final coordinate system of the world (the coordinate system of the root node of the scene). A path through the hierarchy from root to leaf has a corresponding sequence of local transformations whose product is the world transformation for the leaf node. The problem in Figure 4.3 is that the leaf node may be reached via *two paths* through the hierarchy. Each path corresponds to an instance of the leaf object. Realize that the two rooms are placed in different parts of the house. The world transformations applied to the room contents are necessarily different. If you have any plans to make the world transformations persistent, they must be stored *somewhere*. In the tree-based hierarchy, the world transformations are stored directly at the node. To store the world transformations for the DAG of Figure 4.3, you can store them either at each node or in a separate location that the node has pointers to. In either case, a dynamic system is required since the number of parents can be any number and change at any time. World bounding volumes must also be maintained, one per instance.

Another implication is that if you want to change the data directly at the shared node, the Room Contents in our example, it is necessary for you to be able to specify *which instance is to be affected*. This alone creates a complex situation for an application programmer to manage. You may assign a set of names to the shared object, one name per path to the object. The path names can be arbitrarily long, making the use of them a bit overwhelming for the application programmer. Alternatively, you can require that a shared object not be directly accessible. The instances must be managed only through the parent nodes. In our example, to place Room 1 in the House, you set its local transformations accordingly. Room 2 is placed in the world with a different

set of local transformations. The Room Contents always have the identity transformation, never to be changed. This decision has the consequence that if you only have a single instance (most likely the common case in a scene), a parent node should be used to indirectly access that instance. If you are not consistent in the manner of accessing the object, your engine logic must distinguish between a single instance of an object and multiple instances of an object, then handle the situations differently. Thus, every geometric object must be manipulated as a node-geometry pair. Worse is that if you plan on instancing a subgraph of nodes, that subgraph must have parent nodes through which you access the instances. Clearly, this leads to “node bloat” (for lack of a better term), and the performance of updating such a system is not optimal for real-time needs.

Is this speculation or experience? The latter, for sure. One of the first tasks I was assigned when working on NetImmerse in its infancy was to support instancing in the manner described here. Each node stored a dynamic array of parent links and a dynamic array of child links. A corresponding dynamic array of geometric data was also maintained that stored transformations, bounding volumes, and other relevant information. Instances were manipulated through parent nodes, with some access allowed to the instances themselves. On a downward traversal of the scene by a recursive function, the parent pointer was passed to that function and used as a lookup in the child’s parent array to determine which instance the function was to affect. This mechanism addresses the issue discussed earlier, unique names for the paths to the instance. Unfortunately, the system was complicated to build and complicated to maintain (adding new recursive functions for scene traversal was tedious), and the parent pointer lookup was a noticeable time sink, as shown by profiling any applications built on top of the engine. To eliminate the cost of parent pointer lookups, the node class was modified to include an array of instance pointers, one per child of the node. Those pointers were passed through recursive calls, thus avoiding the lookups, and used directly. Of course, this increased the per-node memory requirements and increased the complexity of the system. In the end we decided that supporting instancing by DAGs was not acceptable.

That said, instancing still needs to be supported in an engine. I mentioned this earlier and mention it again: What is important regarding instancing is that (1) you reduce memory usage, and (2) you reduce the artist’s workload. The majority of memory consumption has to do with models with *large* amounts of data. For example, a model with 10,000 vertices, multiple 32-bit texture images, each 512×512 , and corresponding texture coordinates consumes a lot of memory. Instancing such a model will avoid duplication of the large data. The amount of memory that a node or geometry object requires to support core scene graph systems is quite small relative to the actual model data. If a subgraph of nodes is to be instanced, duplication of the nodes requires only a small amount of additional memory. The model data is shared, of course. Wild Magic 4 chooses to share in the manner described here. The sharing is *low level*; that is, instancing of models involves geometric data. If you want to instance an object of a Geometry-derived class, you create two unique Geometry-derived objects, but ask them to share their vertex and index buffers. The

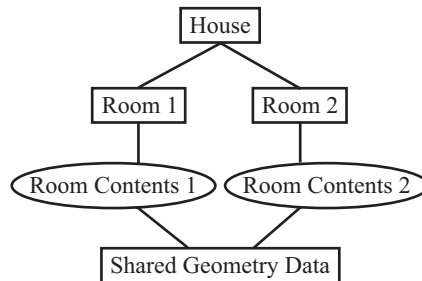


Figure 4.4 The scene graph of Figure 4.3, but with instancing at a low level (geometric data) rather than at a node level.

DAG of Figure 4.3 abstractly becomes the graph shown in Figure 4.4. The work for creating an instance is more than what a DAG-style system requires, but the run-time performance is much improved and the system complexity is minimal.

4.2 GEOMETRIC STATE

Four basic objects involving geometric state are vertex buffers, index buffers, transformations, and bounding volumes.

4.2.1 VERTEX BUFFERS AND INDEX BUFFERS

The vertex positions and attributes, such as normal vectors, colors, and texture coordinates, are all stored together in a *vertex buffer*. There is a one-to-one correspondence between elements of the buffer and vertices in the geometric primitive. The topology for the adjacency information of the vertices is stored in the *index buffer*. The geometric primitive represents a collection of points (the index buffer specifies only in which order the points are drawn), a collection of line segments (i.e., a polyline), a triangle mesh, or various other triangle collections (triangle strips, triangle fans, and so on). The Geometry objects just act as a container for this information so that it is readily available for drawing by the renderer. The vertex buffer and index buffer have counterparts stored in video memory. These are used for the actual drawing. Chapter 3 contains all the details of how this is done.

4.2.2 TRANSFORMATIONS

Wild Magic version 2 supported transformations involving translations $\mathbf{T}$, rotations R , and uniform scaling $\sigma > 0$. A vector $\mathbf{X}$ is transformed to a vector $\mathbf{Y}$ by

$$\mathbf{Y} = R(\sigma\mathbf{X}) + \mathbf{T} \quad (4.1)$$

The order of application is scale first, rotation second, and translation third. However, the order of uniform scaling and rotation is irrelevant. The inverse transformation is

$$\mathbf{X} = \frac{1}{\sigma} R^T(\mathbf{Y} - \mathbf{T}) \quad (4.2)$$

Generally, a graphics API allows for any affine transformation, in particular nonuniform scaling. The natural extension of Equation (4.1) to allow nonuniform scale $S = \text{Diag}(\sigma_0, \sigma_1, \sigma_2)$, $\sigma_i > 0$, for all i , is

$$\mathbf{Y} = R\mathbf{S}\mathbf{X} + \mathbf{T} \quad (4.3)$$

The order of application is scale first, rotation second, and translation third. In this case the order of nonuniform scaling and rotation is relevant. Switching the order produces different results since, in most cases, $RS \neq SR$. The inverse transformation is

$$\mathbf{X} = S^{-1}R^T(\mathbf{Y} - \mathbf{T}) \quad (4.4)$$

where $S^{-1} = \text{Diag}(1/\sigma_0, 1/\sigma_1, 1/\sigma_2)$. The memory requirements to support nonuniform scaling are modest—only two additional floating-point numbers to store.

Wild Magic version 2 disallowed nonuniform scaling because of some undesirable consequences. One goal was to minimize the time spent on matrix and vector arithmetic. This was particularly important when an application had a physical simulation that made heavy use of the transformation system. Using operation counts as a measure of execution time,² let μ represent the number of cycles for a multiplication, let α represent the number of cycles for an addition/subtraction, and let δ represent the number of cycles for a division. On an Intel Pentium class processor, μ and α are equal, both three cycles. The value δ is 39 cycles. Both Equations (4.1) and (4.3) use $12\mu + 9\alpha$ cycles to transform a single vector. Equation (4.2) uses $12\mu + 9\alpha + \delta$ cycles. The only difference between the inverse transform and the forward transform is the division required to compute the reciprocal of scale. The reciprocal is computed first, and then the three components of the vector are multiplied. Equation (4.4) uses

2. A warning about operation counting: Current-day processors have other issues now that can make operation counting an inaccurate measure of performance. You need to pay attention to memory fetches, cache misses, branch penalties, and other architectural aspects.

$9\mu + 9\alpha + 3\delta$ cycles. Unlike the uniform scale inversion, the reciprocals are not computed first. The three vector components are divided directly by the nonuniform scales, leading to three fewer multiplications, but two more divisions. This is still a significant increase in cost because of the occurrence of the additional divisions. The three divisions in $(x/\sigma_0, y/\sigma_1, z/\sigma_2) = (x', y', z')$ may be avoided by instead computing $a = \sigma_0\sigma_1$, $p = a\sigma_2$, $r = 1/p$, $q = r\sigma_2$, $x' = q\sigma_1x$, $y' = q\sigma_0y$, and $z' = raz$. The three divisions cost 3δ cycles, but the alternative costs $9\mu + \delta$ cycles. If the CPU supports a faster but lower-precision division, the increase is not as much of a factor, but you pay in terms of accuracy of the final result. With the advent of specialized hardware such as extended instructions for CPUs, game console hardware, and vector units in general, the performance for nonuniform scaling is not really a concern.

A second issue that is mathematical and one that hardware cannot eliminate is the requirement to factor transformations to maintain the ability to store at each node the scales, the rotation matrix, and the translation vector. To be precise, if you have a path of nodes in a hierarchy and corresponding local transformations, the world transformation is a composition of the local ones. Let the local transformations be represented as homogeneous matrices in block-matrix form. The transformation $\mathbf{Y} = \mathbf{R}\mathbf{S}\mathbf{X} + \mathbf{T}$ is represented by

$$\begin{bmatrix} \mathbf{Y} \\ 1 \end{bmatrix} \begin{bmatrix} \mathbf{RS} & \mathbf{T} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix}$$

The composition of two local transformations $\mathbf{Y} = \mathbf{R}_1\mathbf{S}_1\mathbf{X} + \mathbf{T}_1$ and $\mathbf{Z} = \mathbf{R}_2\mathbf{S}_2\mathbf{Y} + \mathbf{T}_2$ is represented by a homogeneous block matrix that is a product of the two homogeneous block matrices representing the individual transformations:

$$\begin{bmatrix} \mathbf{R}_2\mathbf{S}_2 & \mathbf{T}_2 \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}_1\mathbf{S}_1 & \mathbf{T}_1 \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_2\mathbf{S}_2\mathbf{R}_1\mathbf{S}_1 & \mathbf{R}_2\mathbf{S}_2\mathbf{T}_1 + \mathbf{T}_2 \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{M} & \mathbf{T} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where $\mathbf{M} = \mathbf{R}_2\mathbf{S}_2\mathbf{R}_1\mathbf{S}_1$ and $\mathbf{T} = \mathbf{R}_2\mathbf{S}_2\mathbf{T}_1 + \mathbf{T}_2$. A standard question that is asked somewhat regularly in the Usenet computer graphics newsgroups is how to factor

$$\mathbf{M} = \mathbf{R}\mathbf{S}$$

where $\mathbf{R}$ is a rotation and $\mathbf{S}$ is a diagonal nonuniform scaling matrix. The idea is to have a transformation class that always stores $\mathbf{R}$, $\mathbf{S}$, and $\mathbf{T}$ as individual components, thus allowing direct evaluation of Equations (4.3) and (4.4). Much to the posters' dismay, the unwanted answer is that you cannot always factor $\mathbf{M}$ in this way. In fact, it is not always possible to factor $\mathbf{D}_1\mathbf{R}_1$ into $\mathbf{R}_2\mathbf{D}_2$, where $\mathbf{D}_1$ and $\mathbf{D}_2$ are diagonal matrices and $\mathbf{R}_1$ and $\mathbf{R}_2$ are rotation matrices.

The best you can do is factor $\mathbf{M}$ using *polar decomposition* or *singular value decomposition* ([Hec94, Section III.4]). The polar decomposition is

$$\mathbf{M} = \mathbf{U}\mathbf{A}$$

where U is an orthogonal matrix and A is a symmetric matrix, but not necessarily diagonal. The singular value decomposition is closely related:

$$M = VDW^T$$

where V and W are orthogonal matrices and D is a diagonal matrix. The two factorizations are related by appealing to the eigendecomposition of a symmetric matrix, $A = WDW^T$, where W is orthogonal and D is diagonal. The columns of W are linearly independent eigenvectors of A , and the diagonal elements of D are the eigenvalues (ordered to correspond to the columns of W). It follows that $V = UW$. Implementing either factorization is challenging because the required mathematical machinery is more than what you might expect.

Had I chosen to support nonuniform scaling in Wild Magic *and* wanted a consistent representation of local and world transformations, the factorization issue prevents me from storing a transformation as a triple (R, S, T) , where R is a rotation, S is a diagonal matrix of scales, and T is a translation. One way out of the dilemma is to use a triple for local transformations, but a pair (M, T) for world transformations. The 3×3 matrix M is the composition of rotations and nonuniform scales through a path in the hierarchy. The memory usage for a world transformation is smaller than for a local one, but only one floating-point number less. The cost for a forward transformation $Y = MX + T$ is $9\mu + 9\alpha$, cheaper than for a local transformation. Less memory usage, faster transformation, but the cost is that you have no scaling or rotational information for the world transformation unless you factor into polar form or use the singular value decomposition. Both factorizations are very expensive to compute. The inverse transformation $X = M^{-1}(Y - T)$ operation count is slightly more complicated to determine. Using a cofactor expansion to compute the inverse matrix

$$M^{-1} = \frac{1}{\det(M)} M^{\text{adj}}$$

where $\det(M)$ is the determinant of M and M^{adj} is the adjoint matrix—the transpose of the matrix of cofactors of M . The adjoint has nine entries, each requiring $2\mu + \alpha$ cycles to compute. The determinant is computed from a row of cofactors, using three more multiplications and two more additions, for a total of $3\mu + 2\alpha$ cycles. The reciprocal of the determinant uses δ cycles. Computing the inverse transformation as

$$X = \frac{1}{\det(M)} \left(M^{\text{adj}}(Y - T) \right)$$

requires $33\mu + 20\alpha + \delta$ cycles. This is a very significant increase in cost compared to the $19\mu + 9\alpha + \delta$ cycles used for computing $X = S^{-1}R^T(Y - T)$.

To avoid the increase in cost for matrix inversion, you could alternatively choose a consistent representation where the transformations are stored as 4-tuples of the form (L, S, R, T) , where L and R are rotation matrices, S is a diagonal matrix

of scales, and $\mathbf{T}$ is a translation. Once a world transformation is computed as a composition of local transformations to obtain M and $\mathbf{T}$, you have to factor $M = LDR$ using the singular value decomposition—yet another expensive proposition.

Given the discussion of nonuniform scaling and the performance issues arising from factorization and/or maintaining a consistent representation for transformations, in Wild Magic version 2, I decided to constrain the transformations to use only uniform scaling. I have relaxed the constraint slightly in Wild Magic version 3. The `Spatial` class stores three scale factors, but only the leaf nodes may set these to be nonuniform. But doesn't this introduce all the problems that I just mentioned? Along a path of n nodes, the last node being a geometry leaf node, the world transformation is a composition of $n - 1$ local transformations that have only uniform scale σ_i , $i \geq 2$, and a final local transformation that has nonuniform scales S_1 :

$$\begin{aligned}
 & \left[\begin{array}{c|c} R_n \sigma_n & \mathbf{T}_n \\ \hline \mathbf{0}^T & 1 \end{array} \right] \cdots \left[\begin{array}{c|c} R_2 \sigma_2 & \mathbf{T}_2 \\ \hline \mathbf{0}^T & 1 \end{array} \right] \left[\begin{array}{c|c} R_1 S_1 & \mathbf{T}_1 \\ \hline \mathbf{0}^T & 1 \end{array} \right] \\
 &= \left[\begin{array}{c|c} R' \sigma' & \mathbf{T}' \\ \hline \mathbf{0}^T & 1 \end{array} \right] \left[\begin{array}{c|c} R_1 S_1 & \mathbf{T}_1 \\ \hline \mathbf{0}^T & 1 \end{array} \right] \\
 &= \left[\begin{array}{c|c} (R' R_1)(\sigma' S_1) & R' \sigma' \mathbf{T}_1 + \mathbf{T}' \\ \hline \mathbf{0}^T & 1 \end{array} \right] \\
 &= \left[\begin{array}{c|c} R'' S'' & \mathbf{T}'' \\ \hline \mathbf{0}^T & 1 \end{array} \right]
 \end{aligned}$$

Because of the commutativity of uniform scale and rotation, the product of the first $n - 1$ matrices leads to another matrix of the same form, as shown. The product with the last matrix groups together the rotations and groups together the scales. The final form of the composition is one that does not require a general matrix inverse calculation. I consider the decision to support nonuniform scales only in the `Geometry` class an acceptable compromise between having only uniform scales or having nonuniform scales available at all nodes.

The class that encapsulates the transformations containing translations, rotations, and nonuniform scales is `Transformation`. The default constructor, destructor, and data members are shown next in a partial listing of the class:

```

class Transformation
{
public:
    Transformation ();
    ~Transformation ();

    static const Transformation IDENTITY;

```

```
private:
    Matrix3f m_kMatrix;
    Vector3f m_kTranslate;
    Vector3f m_kScale;
    bool m_bIsIdentity, m_bIsRSMMatrix, m_bIsUniformScale;
};
```

In a moment I will discuss the public interface to the data members. The matrix represents a rotation or a general transformation. The three scale factors are stored as a 3-tuple, but they could just as easily have been stored as three separate floating-point numbers. The class has three additional data members, all Boolean variables. These are considered *hints* to allow for more efficient composition of transformations. The `m_bIsIdentity` hint is set to true when the transformation is the identity. The `m_bIsRSMMatrix` is set to true when `m_kMatrix` is a rotation matrix and `m_kScale` contains nonuniform scales. The `m_bIsUniformScale` hint is set to true when the components of `m_kScale` have a common value. The default constructor creates the identity transformation, where the rotation is the 3×3 identity matrix, the translation is the 3×1 zero vector, and the three scales are all one. The `m_bIsIdentity`, `m_bIsRSMMatrix`, and `m_bIsUniformScale` hints are all set to true. For an application's convenience, the static class member `IDENTITY` stores the identity transformation.

Part of the public interface to access the members is

```
class Transformation
{
public:
    void SetRotate (const Matrix3f& rkRotate);
    const Matrix3f& GetRotate () const;
    void SetMatrix (const Matrix3f& rkMatrix);
    const Matrix3f& GetMatrix () const;
    void SetTranslate (const Vector3f& rkTranslate);
    const Vector3f& GetTranslate () const;
    void SetScale (const Vector3f& rkScale);
    const Vector3f& GetScale () const;
    void SetUniformScale (float fScale);
    float GetUniformScale () const;
};
```

The Set functions have side effects in that each function sets the `m_bIsIdentity` hint to false. The hint is set, even if the final transformation is the identity. For example, calling `SetTranslate` with the zero vector as input will set the hint to false. I made this choice to avoid having to check if the transformation is really the identity after each component is set. The expected case is that the use of Set functions is to make the transformation something *other* than the identity. Even if we were to test for the identity transformation, the test is problematic when floating-point arithmetic is used. An exact comparison of floating-point values is not robust when some of

the values were computed in expressions, the end results of which were produced after a small amount of floating-point round-off error. The `SetScale` function also has the side effect of setting the `m_bIsUniformScale` hint to `false`. As before, the hint is set even if the input scale vector corresponds to uniform scaling. The `SetRotate` function sets the `m_bIsRSMMatrix` hint to `true`. If the hint is `false` when `GetRotate` is called, an assertion is triggered. The `SetMatrix` function sets the `m_bIsRSMMatrix` and `m_bIsUniformScale` hints to `false`. The `Get` functions have no side effects and return the requested components. These functions are `const`, so the components are read-only.

A public accessor function is provided for the convenience of determining the maximum scale associated with a transformation:

```
class Transformation
{
public:
    float GetNorm () const;
};
```

If the matrix is of the form $M = RS$, where R is a rotation and S is a diagonal matrix representing the scales, then `GetNorm` returns $\max\{|\sigma_0|, |\sigma_1|, |\sigma_2|\}$. If M is a general matrix, then `GetNorm` returns the largest sum of absolute values of row entries. That is, if $M = [m_{rc}]$ for $0 \leq r \leq 2$ and $0 \leq c \leq 2$, then `GetNorm` returns

$$\max\{|m_{00}| + |m_{01}| + |m_{02}|, |m_{10}| + |m_{11}| + |m_{12}|, |m_{20}| + |m_{21}| + |m_{22}|\}$$

Given two matrices R and S , a matrix norm $\|\cdot\|$ generally satisfies the inequality $\|RS\| \leq \|R\|\|S\|$, but equality does not have to happen. Thus, it is not necessarily the case that $\|RS\|$ is equal to $\|S\|$, so multiplying R and S first before computing the norm is not guaranteed to give you the same number as the largest magnitude scale. However, the engine uses the matrix norm only to transform a bounding sphere to obtain another bounding sphere. A nonuniform scaling of a sphere produces an ellipsoid, but the engine has no capability to use ellipsoids as bounding volumes. I use the norm as a uniform scale factor to transform the sphere's radius.

Other convenience functions include the ability to tell a transformation to make itself the identity transformation or to make its scales all one:

```
class Transformation
{
public:
    void MakeIdentity ();
    void MakeUnitScale ();
    bool IsIdentity () const;
    bool IsRSMMatrix () const;
    bool IsUniformScale () const;
};
```

The last three functions just return the current values of the hints.

The basic algebraic operations for transformations include application of a transformation to points, application of an inverse transformation to points, and composition of two transformations. The member functions are

```
class Transformation
{
public:
    Vector3f ApplyForward (const Vector3f& rkInput) const;
    void ApplyForward (int iQuantity, const Vector3f* akInput,
        Vector3f* akOutput) const;

    Vector3f ApplyInverse (const Vector3f& rkInput) const;
    void ApplyInverse (int iQuantity, const Vector3f* akInput,
        Vector3f* akOutput) const;

    void Product (const Transformation& rkA,
        const Transformation& rkB,);

    void Inverse (Transformation& rkInverse);
};
```

The first `ApplyForward` and `ApplyInverse` functions apply to single vectors. The second pair of these functions applies to arrays of vectors. If the transformation is $\mathbf{Y} = \mathbf{R}\mathbf{S}\mathbf{X} + \mathbf{T}$ where $\mathbf{R}$ is a rotation matrix, $\mathbf{S}$ is a diagonal scale matrix, and $\mathbf{T}$ is a translation, function `ApplyForward` computes $\mathbf{Y}$ from the input vector(s) $\mathbf{X}$. Function `ApplyInverse` computes $\mathbf{X} = \mathbf{S}^{-1}\mathbf{R}^T(\mathbf{Y} - \mathbf{T})$ from the input vector(s) $\mathbf{Y}$.

The composition of two transformations is performed by the member function `Product`. The name refers to a product of matrices when the transformations are viewed as 4×4 homogeneous matrices. For example,

```
Transformation kA = <some transformation>;
Transformation kB = <some transformation>;
Transformation kC;

// compute C = A*B
kC.Product(kA,kB);

// compute C = B*A, generally not the same as A*B
kC.Product(kB,kA);
```

We will also need to apply inverse transformations to vectors. Notice that earlier I used both the term *points* and the term *vectors*. The two are abstractly different, as discussed in the study of *affine algebra*. A point $\mathbf{P}$ is transformed as

$$\mathbf{P}' = \mathbf{R}\mathbf{S}\mathbf{P} + \mathbf{T}$$

whereas a vector $\mathbf{V}$ is transformed as

$$\mathbf{V}' = R\mathbf{S}\mathbf{V}$$

You can think of the latter equation as the difference of the equations for two transformed points $\mathbf{P}$ and $\mathbf{Q}$:

$$\mathbf{V} = \mathbf{P} - \mathbf{Q}$$

$$\mathbf{P}' = R\mathbf{S}\mathbf{P} + \mathbf{T}$$

$$\mathbf{Q}' = R\mathbf{S}\mathbf{Q} + \mathbf{T}$$

$$\mathbf{V}' = \mathbf{P}' - \mathbf{Q}' = (R\mathbf{S}\mathbf{P} + \mathbf{T}) - (R\mathbf{S}\mathbf{Q} + \mathbf{T}) = R\mathbf{S}(\mathbf{P} - \mathbf{Q}) = R\mathbf{S}\mathbf{V}$$

In terms of homogeneous vectors, the point $\mathbf{P}$ and vector $\mathbf{V}$ are represented by

$$\begin{bmatrix} \mathbf{P} \\ 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \mathbf{V} \\ 0 \end{bmatrix}$$

The corresponding homogeneous transformations are

$$\begin{bmatrix} RS & | & \mathbf{T} \\ \hline \mathbf{0}^T & | & 1 \end{bmatrix} \begin{bmatrix} \mathbf{P} \\ 1 \end{bmatrix} = \begin{bmatrix} R\mathbf{S}\mathbf{P} + \mathbf{T} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{P}' \\ 1 \end{bmatrix} \quad \text{and}$$

$$\begin{bmatrix} RS & | & \mathbf{T} \\ \hline \mathbf{0}^T & | & 1 \end{bmatrix} \begin{bmatrix} \mathbf{V} \\ 0 \end{bmatrix} = \begin{bmatrix} R\mathbf{S}\mathbf{V} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{V}' \\ 0 \end{bmatrix}$$

The inverse transformation of a vector $\mathbf{V}'$ is

$$\mathbf{V} = S^{-1}R^T\mathbf{V}'$$

The member function that supports this operation is

```
class Transformation
{
public:
    Vector3f InvertVector (const Vector3f& rkInput) const;
};
```

Finally, the inverse of the transformation is computed by

```
void Inverse (Transformation& rkInverse);
```

The translation, rotation, and scale components are computed. If $\mathbf{Y} = R\mathbf{S}\mathbf{X} + \mathbf{T}$, the inverse is $\mathbf{X} = S^{-1}R^T(\mathbf{Y} - \mathbf{T})$. The inverse transformation has scale S^{-1} , rotation R^T ,

and translation $-S^{-1}R^T\mathbf{T}$, all stored by `rkInverse`. If you were to use `rkInverse` to transform a point $\mathbf{Y}$, the result would be

$$\mathbf{X} = R^T S^{-1} \mathbf{Y} - S^{-1} R^T \mathbf{T}$$

which is not correct. The storage that `rkInverse` provides is for convenience only. Call the `Inverse` function, access the individual components of `rkInverse`, and then discard `rkInverse`.

The transformation of a plane from model space to world space is also sometimes necessary. Let the model-space plane be

$$\mathbf{N}_0 \cdot \mathbf{X} = c_0$$

where $\mathbf{N}_0$ is a unit-length normal vector, c_0 is a constant, and $\mathbf{X}$ is any point on the plane and is specified in model-space coordinates. The inverse transformation of the point is $\mathbf{X} = S^{-1}R^T(\mathbf{Y} - \mathbf{T})$, where $\mathbf{Y}$ is the point in world-space coordinates. Substituting this in the plane equation leads to

$$\mathbf{N}_1 \cdot \mathbf{Y} = c_1, \quad \mathbf{N}_1 = \frac{RS^{-1}\mathbf{N}_0}{|RS^{-1}\mathbf{N}_0|}, \quad c_1 = \frac{c_0}{|RS^{-1}\mathbf{N}_0|} + \mathbf{N}_1 \cdot \mathbf{T}$$

The member function that supports this operation is

```
class Transformation
{
public:
    Plane3f ApplyForward (const Plane3f& rkInput) const;
};
```

The input plane has normal $\mathbf{N}_0$ and constant c_0 . The output plane has normal $\mathbf{N}_1$ and constant c_1 .

In all the transformation code, I take advantage of the `m_bIsIdentity` and `m_bIsUniformScale` hints. Two prototypical cases are the implementation of `ApplyForward` that maps $\mathbf{Y} = R\mathbf{S}\mathbf{X} + \mathbf{T}$ and the implementation of `ApplyInverse` that maps $\mathbf{X} = S^{-1}R^T(\mathbf{Y} - \mathbf{T})$. The forward transformation implementation is

```
Vector3f Transformation::ApplyForward (
    const Vector3f& rkInput) const
{
    if (m_bIsIdentity)
    {
        // Y = X
        return rkInput;
    }
}
```



```

if (m_bIsRSMatrix)
{
    //  $Y = R*S*X + T$ 
    Vector3f kOutput(
        m_kScale.X()*rkInput.X(),
        m_kScale.Y()*rkInput.Y(),
        m_kScale.Z()*rkInput.Z());
    kOutput = m_kMatrix*kOutput + m_kTranslate;
    return kOutput;
}

//  $Y = M*X + T$ 
Vector3f kOutput = m_kMatrix*rkInput + m_kTranslate;
return kOutput;
}

```

If the transformation is the identity, then $Y = X$ and the output vector is simply the input vector. A generic implementation might do all the matrix and vector operations anyway, not noticing that the transformation is the identity. The hint flag helps avoid those unnecessary calculations. If the transformation is not the identity, it does not matter whether the scale is uniform or nonuniform since three multiplications by a scale parameter occur in either case.

The inverse transformation implementation is

```

Vector3f Transformation::ApplyInverse (
    const Vector3f& rkInput) const
{
    if (m_bIsIdentity)
    {
        //  $X = Y$ 
        return rkInput;
    }

    Vector3f kOutput = rkInput - m_kTranslate;
    if (m_bIsRSMatrix)
    {
        //  $X = S^{-1} * R^t * (Y - T)$ 
        kOutput = kOutput*m_kMatrix;
        if (m_bIsUniformScale)
        {
            kOutput /= GetUniformScale();
        }
        else

```

```

        {
            float fSXY = m_kScale.X()*m_kScale.Y();
            float fSXZ = m_kScale.X()*m_kScale.Z();
            float fSYZ = m_kScale.Y()*m_kScale.Z();
            float fInvDet = 1.0f/(fSXY*m_kScale.Z());
            kOutput.X() *= fInvDet*fSYZ;
            kOutput.Y() *= fInvDet*fSXZ;
            kOutput.Z() *= fInvDet*fSXY;
        }
    }
    else
    {
        // X = M^{-1}*(Y - T)
        kOutput = m_kMatrix.Inverse()*kOutput;
    }

    return kOutput;
}

```

If the transformation is the identity, then $\mathbf{X} = \mathbf{Y}$ and there is no reason to waste cycles by applying the transformation components. Unlike `ApplyForward`, if the transformation is not the identity, then there is a difference in performance between uniform and nonuniform scaling.

For uniform scale, $R^T(\mathbf{Y} - \mathbf{T})$ has all three components divided by scale. The `Matrix3` class has an operator function such that a product of a vector (the left operand $\mathbf{V}$) and a matrix (the right operand M) corresponds to $M^T\mathbf{V}$. The previously displayed code block uses this function. The `Vector3` class supports division of a vector by a scalar. Internally, the reciprocal of the divisor is computed and multiplies the three vector components. This avoids the division occurring three times, replacing the operation instead with a single division and three multiplications.

For nonuniform scale, I use the trick described earlier for avoiding three divisions. The displayed code replaces the three divisions by ten multiplications and one division. For an Intel Pentium that uses 3 cycles per multiplication and 39 cycles per division, the three divisions would cost 78 cycles, but the ten multiplications and one division cost 69 cycles.

4.2.3 BOUNDING VOLUMES

The term *bounding volume* is quite generic and refers to any object that contains some other object. The simplest bounding volumes that game programmers use tend to be spheres or axis-aligned bounding boxes. Slightly more complicated is an oriented bounding box. Yet more complicated is the convex hull of the contained

object, a convex polyhedron. In all cases, the bounding volumes are convex. To be yet more complicated, a bounding volume might be constructed as a union of (convex) bounding volumes.

Culling

One of the major uses for bounding volumes in an engine is for the purpose of *culling* objects. If an object is completely outside the view frustum, there is no reason to tell the renderer to try to draw it, because if the renderer made the attempt, it would find that all triangles in the meshes that represent the object are outside the view frustum. Such a determination does take some time—better to avoid wasting cycles on this, if possible. The scene graph management system could itself determine if the mesh triangles are outside the view frustum, testing them one at a time for intersection with, or containment by, the view frustum, but this gains us nothing. In fact, this is potentially slower when the renderer has a specialized GPU to make the determination, but the scene graph system must rely on a general CPU.

A less aggressive approach is to use a convex bounding volume as an approximation to the region of space that the object occupies. If the bounding volume is outside the view frustum, then so is the object and we need not ask the renderer to draw it. The intersection/containment test between bounding volume and view frustum is hopefully a lot less expensive to compute than the intersection/containment tests for all the triangles of the object. If the bounding volume is a sphere, the test for the sphere being outside the view frustum is equivalent to computing the distance from the sphere center to the view frustum and showing that it is larger than the radius of the sphere.

Computing the distance from a point to a view frustum is more complicated than most game programmers care to deal with—hence the replacement of that test with an *inexact* query that is simpler to implement. Specifically, the sphere is tested against each of the six frustum planes. The frustum plane normals are designed to point into the view frustum; that is, the frustum is on the “positive side” of all the planes. If the sphere is outside any of these planes, say, on the “negative side” of a plane, then the sphere is outside the entire frustum and the object is not visible and therefore not sent to the renderer for drawing (i.e., it is culled). I call this *plane-at-a-time culling*. The geometry query I refer to as the *which-side-of-plane* query. There are situations when the sphere is not outside one of the planes but is outside the view frustum; that is why earlier I used the adjective “inexact.” Figure 2.22 shows the situation in two dimensions.

The sphere in the upper right of the image is not outside any of the frustum planes but is outside the view frustum. The plane-at-a-time culling system determines that the sphere is not outside any plane, and the object associated with the bounding volume is sent to the renderer for drawing. The same idea works for convex bounding volumes other than spheres. Pseudocode for the general inexact culling is

```

bool IsCulled (ViewFrustum frustum, BoundingBox bound)
{
    for each plane of frustum do
    {
        if bound is on the negative side of plane then
            return true;
    }
    return false;
}

```

Hopefully, the occurrence of false positives (bound outside frustum, but not outside all frustum planes) is infrequent.

Even though plane-at-a-time culling is inexact, it may be used to improve efficiency in visibility determination in a scene graph. Consider the scene graph of Figure 4.2, where each node in the tree has a bounding volume associated with it. Suppose that, when testing the bounding volume of the Table Group against the view frustum, you find that the bounding volume is on the positive side of one of the view frustum planes. The collective object represented by Table Group is necessarily on the positive side of that plane. Moreover, the objects represented by the children of Table Group must also be on the positive side of the plane. We may take advantage of this knowledge and pass enough information to the children (during a traversal of the tree for drawing purposes) to let the culling system know not to test the child bounding volumes against that same plane. In our example, the Table and Utensil Group nodes do not have to compare their bounding volumes to that plane of the frustum. The information to be stored is as simple as a bit array, each bit corresponding to a plane. In my implementation, discussed in more detail later in this chapter, the bits are set to 1 if the plane should be compared with the bounding volumes, and 0 otherwise.

An argument I read about somewhat regularly in some Usenet newsgroups is that complicated bounding volumes should be avoided because the which-side-of-plane query for the bounding volume is expensive. The recommendation is to use something as simple as a sphere because the query is very inexpensive to compute compared to, say, an oriented bounding box. Yes, a true statement, but it is taken out of context of the bigger picture. There is a balance between the complexity of the bounding volume type and the cost of the which-side-of-plane query. As a rule of thumb, the more complex the bounding volume of the object, the better fitting it is to the object, but the query is more expensive to compute. Also as a rule of thumb, the better fitting the bounding volume, the more likely it is to be culled compared to a worse-fitting bounding volume. Figure 4.5 shows a typical scenario.

Even though the cost for the which-side-of-plane query is more expensive for the box than for the sphere, the combined cost of the query for the sphere *and* the attempt to draw the object, only to find out it is not visible, is *larger* than the cost of the query for the box. The latter object has no rendering cost because it was culled.

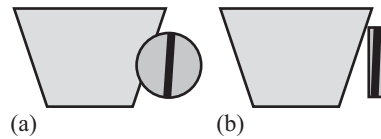


Figure 4.5 A situation where a better-fitting bounding volume leads to culling, but a worse-fitting one does not. (a) The bounding sphere is not tight enough to induce culling. (b) The bounding box is tight enough to induce culling.

On the other hand, if most of the objects are typically inside the frustum, in which case you get the combined cost of the query and drawing, the sphere bounding volumes look more attractive. Whether or not the better-fitting and more expensive bounding volumes are beneficial depends on your specific 3D environment. To be completely certain of which way to go, allow for different bounding volume types and profile your applications for each type to see if there is any savings in time for the better-fitting volumes. The default bounding volume type in Wild Magic is a bounding sphere; however, the system is designed to allow you to easily swap in another type without having to change engine or application code. This is accomplished by providing an abstract interface (base class) for bounding volumes. I discuss this a bit later in the section.

Collision Determination

Another major use for bounding volumes is *3D picking*. A picking ray in world coordinates is selected by some mechanism. A list of objects that are intersected by the ray can be assembled. As a coarse-level test, if the ray does not intersect the bounding volume of an object, then it does not intersect the object.

The bounding volumes also support *collision determination*. More precisely, they may be used to determine if two objects are *not intersecting*, much in the same way they are used to determine if an object is not visible. Collision detection for two arbitrary triangle meshes is an expensive endeavor. We use a bounding volume as an approximation to the region of space that the object occupies. If the bounding volumes of two objects do not intersect, then the objects do not intersect. The hope is that the test for intersection of two bounding volumes is much less expensive than the test for intersection of two triangle meshes. Well, it is, unless the objects themselves are single triangles!

The discussion of how to proceed with picking after you find out that the ray intersects a bounding volume or how to proceed with collision detection after you find out that the bounding volumes intersect is deferred to Section 8.4.

The Abstract Bounding Volume Interface

My main goal in having an abstract interface was not to force the engine users to use my default bounding spheres. I also wanted to make sure that it was very easy to make the change, one that did not require changes to the core engine components or the applications themselves. The abstraction forces you to think about the various geometric queries in object-independent ways. Although abstract interfaces tend not to have data associated with them, experience led me to conclude that a minimal amount of information is needed. At the lowest level, you need to know *where* a bounding volume is located and what its *size* is. The two data members that represent these values are a center point and a radius. These values already define a sphere, so you may think of the base class as a representation of a bounding sphere for the bounding volume. The values for an oriented bounding box are naturally the box center and the maximum distance from the center to a vertex. The values for a convex polyhedron may be selected as the average of the vertices and the maximum distance from that average to any vertex. Other types of bounding volumes can define center and radius similarly.

The abstract class is `BoundingBox` and has the following initial skeleton:

```
class BoundingBox : public Object
public:
{
    virtual ~BoundingBox ();

    virtual void SetCenter (const Vector3f& rkCenter) = 0;
    virtual void SetRadius (float fRadius) = 0;
    virtual Vector3f GetCenter () const = 0;
    virtual float GetRadius () const = 0;

    static BoundingBox* Create ();

protected:
    BoundingBox ();
};
```

The constructor is protected and the Set and Get accessors are pure virtual functions, so the class is abstract. The static member function `Create` is used as a factory to produce objects without having to know what specific type (or types) exist in the engine. A derived class has the responsibility for implementing this function, and only one derived class may do so. In the engine, the `Create` call occurs during construction of a `Spatial` object (the world bounding volume) and a `Geometry` object (the model bounding volume). A couple of additional calls occur in `Geometry`-derived classes,

but only because the construction of the model bounding volume is deferred until the actual model data is known by those classes.

Even though only a single derived class implements `Create`, you may have multiple `BoundingBoxVolume`-derived classes in the engine. Those not implementing `Create` must be constructed explicitly. Only the core engine components for geometric updates must be ignorant of the type of bounding volume.

Switching to a new `BoundingBoxVolume` type for the core engine is quite easy. All you need to do is comment out the implementation of `BoundingBoxVolume::Create` in the default bounding volume class, `SphereBV`, and implement it in your own derived class. The `SphereBV` class is

```
BoundingBoxVolume* BoundingBoxVolume::Create ()
{
    return new SphereBV;
}
```

If you were to switch to `BoxBV`, the oriented bounding box volumes, then in `Wm4BoxBV.cpp` you would place

```
BoundingBoxVolume* BoundingBoxVolume::Create ()
{
    return new BoxBV;
}
```

The remaining interface for `BoundingBoxVolume` is shown next. All member functions are pure virtual, so the derived classes must implement these.

```
class BoundingBoxVolume : public Object
public:
{
    virtual void ComputeFromData (
        const Vector3fArray* pkVertices) = 0;

    virtual void TransformBy (const Transformation& rkTransform,
        BoundingBoxVolume* pkResult) = 0;

    virtual int WhichSide (const Plane3f& rkPlane) const = 0;

    virtual bool TestIntersection (const Vector3f& rkOrigin,
        const Vector3f& rkDirection) const = 0;

    virtual bool TestIntersection (
        const BoundingBoxVolume* pkInput) const = 0;
```

```

    virtual void CopyFrom (const BoundingBox* pkInput) = 0;
    virtual void GrowToContain (const BoundingBox* pkInput) = 0;
    virtual bool Contains (const Vector3f& rkPoint) const = 0;
};

```

The bounding volume depends, of course, on the vertex data that defines the object. The `ComputeFromData` method provides the construction of the bounding volume from the vertices.

The transformation of a model-space bounding volume to one in world space is supported by the method `TransformBy`. The first input is the model-to-world transformation, and the second input is the world-space bounding volume. That volume is computed by the method and is valid on return from the function. The `Geometry` class makes use of this function.

The method `WhichSide` supports the which-side-of-plane query that was discussed for culling of nonvisible objects. The `Plane3` class stores unit-length normal vectors, so the `BoundingBox`-derived classes may take advantage of that fact to implement the query. If the bounding volume is fully on the positive side of the plane (the side to which the normal points), the function returns $+1$. If it is fully on the negative side, the function returns -1 . If it straddles the plane, the function returns 0 .

The first `TestIntersection` method supports 3D picking. The input is the origin and direction vector for a ray that is in the same coordinate system as the bounding volume. The direction vector must be unit length. The return value is `true` if and only if the ray intersects the bounding volume. The second `TestIntersection` method supports collision determination. The input bounding volume must be the same type as the calling object, but the engine does not check this constraint, so you must. The bounding volumes are assumed to be stationary. The return value of the function is `true` if and only if the two bounding volumes are intersecting.

The member functions `CopyFrom` and `GrowToContain` support the upward pass through the scene graph that computes the bounding volume of a parent node from the bounding volumes of the child nodes. In *Wild Magic*, the parent bounding volume is constructed to contain all the child bounding volumes. The default bounding volume is a sphere, so the parent bounding volume is a sphere that contains all the spheres of the children. The function `CopyFrom` makes the calling object a copy of the input bounding volume. The function `GrowToContain` constructs the bounding volume of the calling bounding volume and the input bounding volume. For a node with multiple children, `CopyFrom` makes a copy of the first child, and `GrowToContain` creates a bounding volume that contains that copy and the bounding volume of the second child. The resulting bounding volume is grown further to contain each of the remaining children. The member function `Contains` is a simple test of whether the input point is inside the bounding volume.

A brief warning about having a bounding volume stored in `Spatial` through an abstract base class (smart) pointer: Nothing prevents you from setting the bounding volume of one object to be a sphere and another to be a box. However, the `BoundingBox` member functions that take a `BoundingBox` object as input are

designed to manipulate the input as if it were the same type as the calling object. Mixing bounding volume types is therefore an error, and the engine has no prevention mechanism for this. It is possible to extend the bounding volume system to handle mixed types. Each bounding volume pair requires you to implement a test-intersection query function. The semantics of the `CopyFrom` function must change. How do you copy a bounding sphere to an oriented bounding box? The semantics of `GrowToContain` must also change. If you have a sphere and a box, should the containing volume be a sphere or a box? I chose to limit the complexity of Wild Magic by disallowing mixing of bounding volume types.

Another warning is that the merging of bounding volumes two at a time is a greedy algorithm. The final bounding volume is not usually optimal, leading to less precise culling. A joint merge of bounding volumes will usually produce a better fit, but the run-time cost of a joint merge is typically more than that for a greedy algorithm. This is a trade-off you will need to consider when designing a hierarchical culling system.

4.2.4 GEOMETRIC TYPES

The basic geometric types supported in the engine are collections of points, collections of line segments, triangle meshes, and particles. Various classes in the core engine implement these types. During the drawing pass through the scene graph, the renderer is provided with such objects and must draw them as their types dictate. Most graphics APIs require the type of object to be specified, usually via a set of enumerated values. To facilitate this, the `Geometry` class has enumerations for the basic types, as shown in the following code snippet:

```
class Geometry : public Spatial
{
    // internal use
    public:
        enum GeometryType
        {
            GT_POLYPOINT,
            GT_POLYLINE_SEGMENTS,
            GT_POLYLINE_OPEN,
            GT_POLYLINE_CLOSED,
            GT_TRIMESH,
            GT_MAX_QUANTITY
        };

        GeometryType Type;
};
```

The type itself is stored in the data member `Type`. It is in public scope because there are no side effects in reading or writing it. However, the block is marked for internal use by the engine. There is no need for an application writer to manipulate the type.

The value `GT_POLYPOINT` indicates that the object is a collection of points. The value `GT_TRIMESH` indicates that the object is a triangle mesh. The three values with `POLYLINE` as part of their names are used for collections of line segments. `GT_POLYLINE_SEGMENTS` is for a set of line segments with no connections between them. `GT_POLYLINE_OPEN` is for a polyline, a set of line segments where each segment endpoint is shared by at most two lines. The initial and final segments each have an endpoint that is not shared by any other line segment; thus, the polyline is said to be *open*. Another term for an open polyline is a *line strip*. If the two endpoints are actually the same point, then the polyline forms a loop and is said to be *closed*. Another term for a closed polyline is a *line loop*.

If you were to modify the engine to support other types that are native to the graphics APIs, you could add enumerated types to the list. You should add these after `GT_TRIMESH`, but before `GT_MAX_QUANTITY`, in order to preserve the numeric values of the current types.

Points

A collection of points is represented by the class `Polypoint`, which is derived from `Geometry`. The interface is very simple.

```
class Polypoint : public Geometry
{
public:
    Polypoint (VertexBuffer* pkVBuffer);
    virtual ~Polypoint ();

    void SetActiveQuantity (int iActiveQuantity);
    int GetActiveQuantity () const;

protected:
    Polypoint ();

    int m_iActiveQuantity;
};
```

The points are provided to the constructor. From the application's perspective, the set of points is unordered. However, for the graphics APIs that use vertex arrays, I have chosen to assign indices to the points. The vertices and indices are both used for drawing. The public constructor is

```

Polypoint::Polypoint (VertexBuffer* pkVBuffer)
:
    Geometry(pkVBuffer,0)
{
    Type = GT_POLYPOINT;

    int iVQuantity = VBuffer->GetVertexQuantity();
    m_iActiveQuantity = iVQuantity;

    IBuffer = WM4_NEW IndexBuffer(iVQuantity);
    int* aiIndex = IBuffer->GetData();
    for (int i = 0; i < iVQuantity; i++)
    {
        aiIndex[i] = i;
    }
}

```

The assigned indices are the natural ones.

The use of an index array has a pleasant consequence. Normally, all of the points would be drawn by the renderer. In some applications you might want to have storage for a large collection of points, but have only a subset *active* at one time. The class has a data member, `m_iActiveQuantity`, that indicates how many are active. The active quantity may be zero but cannot be larger than the total quantity of points. The active set is contiguous in the array, starting at index zero, but if need be, an application can move the points from one vertex array location to another.

The active quantity data member is not in the public interface. The function `SetActiveQuantity` has the side effect of validating the requested quantity. If the input quantity is invalid, the active quantity is set to the total quantity of points.

The index buffer `IBuffer` is a data member in the base class `Geometry`. Its type is `IndexBuffer`. This array is used by the renderer for drawing purposes. Part of that process involves querying the array for the number of elements. The index buffer class has a member function, `GetIndexQuantity`, that returns the *total* number of elements in the array. However, we want it to report the active quantity when the object to be drawn is of type `Polypoint`. To support this, the shared array class has a member function `SetIndexQuantity` that changes the internally stored total quantity to the requested quantity. The requested quantity must be no larger than the original total quantity. If it is not, no reallocation occurs in the shared array, and any attempt to write elements outside the original array is an access violation.

Rather than adding a new data member to `IndexBuffer` to store an active quantity, allowing the total quantity to be stored at the same time, I made the decision that the caller of `SetIndexQuantity` must remember the original total quantity, in case the original value must be restored through another call to `SetIndexQuantity`. My decision is based on the observation that calls to `SetIndexQuantity` will be infrequent, so I wanted to minimize the memory usage for the data members of `IndexBuffer`.

As in all Object-derived classes, a default constructor is provided for the purpose of streaming. The constructor is protected to prevent the application from creating default objects whose data members have not been initialized with real data.

Line Segments

A collection of line segments is represented by the class `Polyline`, which is derived from `Geometry`. The interface is

```
class Polyline : public Geometry
{
public:
    Polyline (VertexBuffer* pkVBuffer, bool bClosed,
              bool bContiguous);
    virtual ~Polyline ();

    void SetActiveQuantity (int iActiveQuantity);
    int GetActiveQuantity () const;
    void SetClosed (bool bClosed);
    bool GetClosed () const;
    void SetContiguous (bool bContiguous);
    bool GetContiguous () const;

protected:
    Polyline ();
    void SetGeometryType ();

    int m_iActiveQuantity;
    bool m_bClosed, m_bContiguous;
};
```

The endpoints of the line segments are provided to the constructor. The three possible interpretations for the vertices are disjoint segments, open polyline, or closed polyline. The input parameters `bClosed` and `bContiguous` determine which interpretation is used. The inputs are stored as class members `m_bClosed` and `m_bContiguous`. The actual interpretation is implemented in `SetGeometryType`:

```
void Polyline::SetGeometryType ()
{
    if (m_bContiguous)
    {
        if (m_bClosed)
        {
```

```

    if (Type != GT_POLYLINE_CLOSED)
    {
        // Increase the index quantity to account
        // for closing the polyline.
        IBuffer->SetIndexQuantity(
            IBuffer->GetIndexQuantity()+1);
        IBuffer->ReleaseAll();
    }
    Type = GT_POLYLINE_CLOSED;
}
else
{
    if (Type == GT_POLYLINE_CLOSED)
    {
        // Decrease the index quantity to account
        // for closing the polyline.
        IBuffer->SetIndexQuantity(
            IBuffer->GetIndexQuantity()-1);
        IBuffer->ReleaseAll();
    }
    Type = GT_POLYLINE_OPEN;
}
}
else
{
    if (Type == GT_POLYLINE_CLOSED)
    {
        // Decrease the index quantity to account
        // for closing the polyline.
        IBuffer->SetIndexQuantity(
            IBuffer->GetIndexQuantity()-1);
        IBuffer->ReleaseAll();
    }
    Type = GT_POLYLINE_SEGMENTS;
}
}

```

To be a polyline where endpoints are shared, the contiguous flag must be set to true. The closed flag has the obvious interpretation.

Let the points be P_i for $0 \leq i < n$. If the contiguous flag is false, the object is a collection of disjoint segments. For a properly formed collection, the quantity of vertices n should be even. The $n/2$ segments are

$$\langle P_0, P_1 \rangle, \langle P_2, P_3 \rangle, \dots, \langle P_{n-2}, P_{n-1} \rangle$$

If the contiguous flag is true and the closed flag is false, the points represent an open polyline with $n - 1$ segments:

$$\langle \mathbf{P}_0, \mathbf{P}_1 \rangle, \langle \mathbf{P}_1, \mathbf{P}_2 \rangle, \dots, \langle \mathbf{P}_{n-2}, \mathbf{P}_{n-1} \rangle$$

The endpoint $\mathbf{P}_0$ of the initial segment and the endpoint $\mathbf{P}_{n-1}$ of the final segment are not shared by any other segments. If instead the closed flag is true, the points represent a closed polyline with n segments:

$$\langle \mathbf{P}_0, \mathbf{P}_1 \rangle, \langle \mathbf{P}_1, \mathbf{P}_2 \rangle, \dots, \langle \mathbf{P}_{n-2}, \mathbf{P}_{n-1} \rangle, \langle \mathbf{P}_{n-1}, \mathbf{P}_0 \rangle$$

Each point is shared by exactly two segments. Although you might imagine that a closed polyline in the plane is a single loop that is topologically equivalent to a circle, you can obtain more complicated topologies by duplicating points. For example, you can generate a bowtie (two closed loops) in the $z = 0$ plane with $\mathbf{P}_0 = (0, 0, 0)$, $\mathbf{P}_1 = (1, 0, 0)$, $\mathbf{P}_2 = (0, 1, 0)$, $\mathbf{P}_3 = (0, 0, 0)$, $\mathbf{P}_4 = (0, -1, 0)$, and $\mathbf{P}_5 = (-1, 0, 0)$. The contiguous and closed flags are both set to true.

The class has the ability to select an active quantity of endpoints that is smaller or equal to the total number, and the mechanism is exactly the one used in `Polypoint`. If your `Polyline` object represents a collection of disjoint segments, you should also make sure the active quantity is an even number.

Triangle Meshes

The simplest representation for a collection of triangles is as a list of m triples of $3m$ vertices:

$$\langle \mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2 \rangle, \langle \mathbf{V}_3, \mathbf{V}_4, \mathbf{V}_5 \rangle, \dots, \langle \mathbf{V}_{3m-3}, \mathbf{V}_{3m-2}, \mathbf{V}_{3m-1} \rangle$$

The vertices of each triangle are listed in counterclockwise order; that is, the triangle is in a plane with a specified normal vector. An observer on the side of the plane to which the normal is directed sees the vertices of the triangle in a counterclockwise order on that plane. A collection like this is sometimes called a *triangle soup* (or more generally, a *polygon soup*). Graphics APIs do support rendering where the triangles are provided this way, but most geometric models built from triangles are not built as a triangle soup. Vertices in the model tend to be part of more than one triangle. Moreover, if the triangle soup is sent to the renderer, each vertex must be transformed from model space to world space, including running them through the clipping and lighting portions of the system. If a point occurs multiple times in the list of vertices, each one processed by the renderer, we are wasting a lot of cycles.

A more efficient representation for a collection of triangles is to have an array of unique vertices and represent the triangles as a collection of triples of indices into the vertex array. This is called a *triangle mesh*. If $\mathbf{V}_i$ for $0 \leq i < n$ is the array of vertices,

an index array I_j for $0 \leq j < 3m$ represents the triangles

$$\langle \mathbf{V}_{I_0}, \mathbf{V}_{I_1}, \mathbf{V}_{I_2} \rangle, \langle \mathbf{V}_{I_3}, \mathbf{V}_{I_4}, \mathbf{V}_{I_5} \rangle, \dots, \langle \mathbf{V}_{I_{3m-3}}, \mathbf{V}_{I_{3m-2}}, \mathbf{V}_{I_{3m-1}} \rangle$$

The goal, of course, is that n is a lot smaller than $3m$ because of the avoidance of duplicate vertices in the vertex array. Fewer vertices must be processed by the renderer, leading to faster drawing.

The class that represents triangle meshes is `TriMesh`. A portion of the interface is

```
class TriMesh : public Geometry
{
public:
    TriMesh (VertexBuffer* pkVBuffer, IndexBuffer* pkIBuffer);
    virtual ~TriMesh ();

    int GetTriangleQuantity () const;
    void GenerateNormals ();

protected:
    TriMesh ();
    virtual void UpdateModelNormals ();
};
```

I have omitted the interface that supports the picking system and will discuss that in Section 8.4.

The constructor requires you to provide the vertex and index buffers for the triangle mesh. The quantity of elements in the index buffer should be a multiple of three. The member function `GetTriangleQuantity` returns the quantity of indices divided by three.

If vertex normals are provided by the input vertex buffer, they will be used as needed during rendering. If they are not provided and you want to add them, you can do so by a call to `GenerateNormals`. This will re-create the vertex buffer with additional storage for the normals. The actual construction of the vertex normals is done in the method `UpdateModelNormals`. The method is protected, so you cannot call it directly. It is called indirectly through the public update function `Geometry::UpdateMS`. Multiple algorithms exist for the construction of vertex normals. The one I implemented is as follows. Let T_1 through T_m be those triangles that share vertex $\mathbf{V}$. Let $\mathbf{N}_1$ through $\mathbf{N}_m$ be normal vectors to the triangles, but not necessarily unit-length ones. For a triangle T with vertices $\mathbf{V}_0$, $\mathbf{V}_1$, and $\mathbf{V}_2$, the normal I use is $\mathbf{N} = (\mathbf{V}_1 - \mathbf{V}_0) \times (\mathbf{V}_2 - \mathbf{V}_0)$. The vertex normal is a unit-length vector,

$$\mathbf{N} = \frac{\sum_{i=1}^m \mathbf{N}_i}{\left| \sum_{i=1}^m \mathbf{N}_i \right|}$$

The length $|N_i|$ is twice the area of the triangle to which it is normal. Therefore, large triangles will have a greater effect on the vertex normal than small triangles. I consider this a more reasonable algorithm than one that computes the vertex normal as an average of unit-length normals for the sharing triangles, where all triangles have the same influence on the outcome regardless of their areas.

Particles

A *particle* is considered to be a geometric primitive with a *location* in space and a *size*. The size attribute distinguishes particles from points. A collection of particles is referred to as a *particle system*. Particle systems are quite useful, for interesting visual displays as well as for physical simulations. In this section I will discuss the geometric aspects of particles and the class `Particles` that represents them.

The portion of the class interface for `Particles` that is relevant to data management is

```
class Particles : public TriMesh
{
public:
    Particles (const Attributes& rkAttr,
               Vector3fArray* pkLocations, FloatArray* pkSizes);
    virtual ~Particles ();

    Vector3fArrayPtr Locations;
    FloatArrayPtr Sizes;
    float SizeAdjust;

    void SetActiveQuantity (int iActiveQuantity);
    int GetActiveQuantity () const;

protected:
    Particles ();
    void GenerateParticles (const Camera* pkCamera);

    int m_iActiveQuantity;
};
```

The first observation is that the class is derived from `TriMesh`. The particles are drawn as billboard squares (see Section 7.1) that always face the observer. Each square is built of two triangles, and all the triangles are stored in the base class as a triangle mesh. The triangle mesh has four times the number of vertices as it does particle locations, which is why the locations are stored as a separate array.

The constructor accepts inputs for the particle's vertex attributes, locations, and sizes. The vertex buffer is created to store four times the number of locations.

The data members `Locations`, `Sizes`, and `SizeAdjust` are in public scope because no side effects must occur when they are read or written. The locations and sizes are as described previously. The data member `SizeAdjust` is used to uniformly scale the particle sizes, if so desired. The adjustment is a multiplier of the sizes stored in the member array `Sizes`, not a replacement for those values. The initial value for the size adjustment is 1.

The class has the ability to select an active quantity of endpoints that is smaller or equal to the total number. The mechanism is exactly the one used in `Polypoint`.

4.3 RENDER STATE

I use the term *render state* to refer to all the information that is associated with the geometric data for the purpose of drawing the objects. Three main categories of render state are *global state*, *lights*, and *effects*.

4.3.1 GLOBAL STATE

Global state refers to information that is essentially independent of any information the objects might provide. The states I have included in the engine are alpha blending, triangle culling, material, polygon offset, stenciling, wireframe, and depth buffering. For example, depth buffering does not care how many vertices or triangles an object has. A material has attributes that are applied to the vertices of an object, regardless of how many vertices it has. Alpha blending is used for combining pixel colors at the color buffer level. A global state, when attached to an interior node in a scene hierarchy, affects all leaf nodes in the subtree rooted at the node. This property is why I used the adjective *global*.

The base class is `GlobalState` and has the following interface:

```
class GlobalState : public Object
{
public:
    virtual ~GlobalState ();

    enum StateType
    {
        ALPHA,
        CULL,
        MATERIAL,
        POLYGONOFFSET,
        STENCIL,
```

```

        WIREFRAME,
        ZBUFFER,
        MAX_STATE_TYPE
    };

    virtual StateType GetStateType () const = 0;

    // default states
    static Pointer<GlobalState> Default[MAX_STATE_TYPE];

protected:
    GlobalState ();
};

```

The base class is abstract since the constructor is protected (and since there is a pure virtual function declared). To support fast access of global states in arrays of smart pointers `GlobalStatePtr`, I chose to avoid using the `Object` run-time type information (RTTI) system. The enumerated type of `GlobalState` provides an alternate RTTI system. Each derived class returns its enumerated value through an implementation of `GetStateType`. Each derived class is also responsible for creating a default state, stored in the static array `GlobalState::Default[]`. Currently, the enumerated values are a list of all the global states I support. If you were to add another one, you would derive a class `MyNewGlobalState` from `GlobalState`. But you also have to add another enumerated value `MYNEWGLOBALSTATE` to the base class. This violates the open-closed principle of object-oriented programming, but the changes to `GlobalState` are so simple and so infrequent that I felt justified in the violation. None of the classes in Wild Magic version 4 ever write an array of global state pointers to disk, so adding a new state does not invalidate all of the scenes you had streamed before the change.

The global states are stored in class `Spatial`. A portion of the interface relative to global state storing and member accessing is

```

class Spatial : public Object
{
public:
    int GetGlobalStateQuantity () const;
    GlobalState* GetGlobalState (int i) const;
    GlobalState* GetGlobalState (GlobalState::StateType eType) const;
    void AttachGlobalState (GlobalState* pkState);
    void DetachGlobalState (GlobalState::StateType eType);
    void DetachAllGlobalStates ();

protected:
    std::vector<GlobalStatePtr> m_kGlobalStates;
};

```

The states are stored in an array of smart pointers. A typical scene will have only a small number of nodes with global states attached, since these states are designed to override the default ones. The names of the member functions make it clear how to use the functions. The `eType` input is intended to be one of the `GlobalState` enumerated values. For example, the code

```
MaterialState* pkMS = <some material state>;
Spatial* pkSpatial = <some Spatial-derived object>;
pkSpatial->AttachGlobalState(pkMS);
pkSpatial->DetachGlobalState(GlobalState::MATERIAL);
```

attaches a material state to an object, then removes it from the object.

The class `Geometry` also has storage for global states, but the storage is for all global states encountered along the path in the scene hierarchy from the root node to the geometry leaf node. The storage is assembled during a render-state update, a topic discussed later in this section. The portion of the interface of `Geometry` relevant to storage is

```
class Geometry : public Spatial
{
// internal use
public:
    GlobalStatePtr States[GlobalState::MAX_STATE_TYPE];
};
```

The array of states is in public scope but is tagged for internal use only. An application should not manipulate the array or its members.

These classes derived from `GlobalState` have a few things in common. First, they must all implement the virtual function `GetStateType`. Second, they must all create default objects, something that is done at program initialization. At program termination, the classes should all destroy their default objects. The initialization-termination system discussed in Section 18.8 is used to perform these actions. You will see that each derived class uses the macros defined in `Wm4Main.mcr` and implements `void Initialize()` and `void Terminate()`. All the derived classes have a default constructor that is used to create the default objects.

The influence of the global states on rendering was discussed in Sections 2.6 and 3.1.

4.3.2 LIGHTS

Drawing objects using only textures results in renderings that lack the realism we are used to in the real world. Much of the richness our own visual systems provide is due to *lighting*. A graphics system must support the concept of lights and of materials that

the lights affect. The lighting models supported by standard graphics APIs are a simple approximation to true lighting but are designed so that the lighting calculations can be performed quickly. More realistic lighting is found in systems that are almost never real time. The physical model aspects of lights and materials were discussed in detail in Section 2.6.2. The discussion in this section is about the design choices for representing lights.

The types of lights and their attributes are sufficiently numerous that many engines provide multiple classes. Usually an engine will provide an abstract base class for lights, and then derived classes such as an ambient light class, a directional light class, a point light class, and a spotlight class. I did so in Wild Magic version 2, but decided that the way the renderer accessed the information for a light was more complicated than it needed to be. Also in Wild Magic version 2, the `Light` class was derived from `Object`. A number of users were critical of this choice and insisted that `Light` be derived from `Spatial`. By doing so, a light automatically has a location (the local translation) and an orientation (the local rotation). One of the orientation vectors can assume the role of the direction for a directional light.

In Wild Magic version 2, I had chosen not to derive `Light` from `Spatial` because ambient lights have no location or direction and directional lights have no location. In this sense they are not very spatial! The consequence, though, was that I had to add a class, `LightNode`, that was derived from `Node` and that had a `Light` data member. This allowed point lights and spotlights to change location and directional lights to change orientation, and then have the geometric update system automatically process them. Even these classes presented some problems to users. One problem had to do with importing `LightWave` objects into the engine, because `LightWave` uses left-handed coordinates for everything. The design of `LightNode` (and `CameraNode`) prevented a correct import of lights (and cameras) when they were to be attached as nodes in a scene.

In the end, I decided to satisfy the users and designed Wild Magic version 3 to create a single class called `Light` that is derived from `Spatial`. Not all data members made sense for each light type. When you manipulated a directional light, setting the location had no effect. By deriving from `Spatial`, some subsystems were available to `Light` that are irrelevant. For example, attaching to a light a global state such as a depth buffer had no meaning, but the engine semantics allowed the attachment. In fact, you could even attach lights to lights. You could attach a light as a leaf node in the scene hierarchy. For example, you might have a representation of a headlight in an automobile. A node is built with two children: One child is the `Geometry` object that represents the headlight's model data, and the other child is a `Light` to represent the light source for the headlight. The geometric data is intended to be drawn to visualize the headlight, but the light object itself is not renderable. The virtual functions for global-state updates and for drawing were stubbed out in the Wild Magic version 2 `Light` class, so that incorrect use of the lights would not be a problem.

These design choices appear to have caused just as much grumbling about the consequences, with one poster to a game developer forum calling the design “ugly.”

For the umpteenth time, I have reversed my design decisions, but this time is for good. Here is my explanation, for better or worse. The `Light` class is no longer derived from `Spatial`; the `Camera` class similarly is no longer derived from `Spatial`. And I revived the classes `LightNode` and `CameraNode`, but I believe I have made sufficient changes to `Light` and `Camera` to avoid the importing problems caused by modeling packages with fully left-handed coordinate systems.

If the `Light` class were to have no data, or just ambient color and intensity, you could use a standard class hierarchy:

```
class Light
    [ambient, intensity]

class AmbientLight : public Light
    [no additional data]

class DirectionalLight : public Light
    [direction, diffuse, specular]

class PointLight : public Light
    [position, diffuse, specular, attenuation]

class SpotLight : public PointLight
    [cone axis, cone angle, spot exponent]
```

The renderer holds onto lights via the base class `Light`. The consequences of a standard class hierarchy are that the renderer must use dynamic casting to determine the type of light in order to set shader program constants in the `Renderer::SetConstantLightFOOBAR` calls. This is an expense I wish to avoid. An alternative is to allow `Light` to store all the data in public scope, but to derive the specific light classes using a *protected* `Light` base class. Thus, `Renderer` has access to all the data it needs without having to dynamically cast, and the derived-class objects have functions to access only the data relevant to them. Unfortunately, you run into problems with access rights to `Object` items such as incrementing and decrementing reference counts for smart pointers. In the end, I chose to make the `Light` class a generic class that stores everything needed by the various light types. I find this a reasonable trade-off, allowing the rapid setting of shader constants to be the important issue.

The Light Class

The `Light` class has a quite complicated interface. I will look at portions of it at a time. The class supports the standard light types: ambient, directional, point, and spot.

```

class Light : public Object
{
public:
    enum LightType
    {
        LT_AMBIENT,
        LT_DIRECTIONAL,
        LT_POINT,
        LT_SPOT,
        LT_QUANTITY
    };

    Light (LightType eType = LT_AMBIENT);
    virtual ~Light ();

    LightType Type;        // default: LT_AMBIENT

    ColorRGB Ambient;      // default: ColorRGB(0,0,0)
    ColorRGB Diffuse;      // default: ColorRGB(0,0,0)
    ColorRGB Specular;     // default: ColorRGB(0,0,0)

    float Constant;        // default: 1
    float Linear;          // default: 0
    float Quadratic;       // default: 0
    float Intensity;       // default: 1

    float Angle;           // default: pi
    float CosAngle;        // default: -1
    float SinAngle;        // default: 0
    float Exponent;        // default: 1
    void SetAngle (float fAngle);

    Vector3f Position, DVector, UVector, RVector;
    void SetDirection (const Vector3f& rkDirection,
                      bool bUnitLength = true);

    bool IsValidFrame () const;
};

```

When you create a light, you specify the type you want. Each light has ambient, diffuse, and specular colors. The data members Constant, Linear, Quadratic, and Intensity are used for attenuation as described in Section 2.6.2.

Spotlights require you to specify an angle from the spotlight axis. This angle is represented by the member Angle. For the convenience and efficiency of shader

programs that use spotlights, you also specify the sine and cosine of the angle. This avoids the vertex program from having to compute it for each vertex. The member `Exponent` is the spotlight exponent; see Section 2.6.2.

The coordinate frame for a light is specified by `Position`, the light position; `DVector`, the light direction; `UVector`, an up vector; and `RVector`, a right vector perpendicular to the other two vectors. The three vectors must form a right-handed orthonormal set. These quantities are what were automatically provided in Wild Magic version 3 by the `Spatial` base class. As mentioned previously, not all light types require all the coordinate frame information. In fact, the current light types supported by the engine use only position and direction vectors, but it is conceivable that you might add a light type that does need an up and a right vector (e.g., a fluorescent light in the shape of a cylinder). The `SetDirection` function automatically computes an up and right vector. These are actually used by the `LightNode` class.

The function `IsValidFrame` is for debugging purposes. It checks to see if the coordinate frame vectors really do form a right-handed orthonormal set.

Support for Lights in Spatial and Geometry Classes

The `Spatial` class stores an array of lights. If a light is added to this array, and the object really is of type `Node`, then my design choice is that the light illuminates all leaf geometry in the subtree rooted at the node. The portion of the interface for `Spatial` relevant to adding and removing lights from the list is

```
class Spatial : public Object
{
public:
    int GetLightQuantity () const;
    Light* GetLight (int i) const;
    void AttachLight (Light* pkLight);
    void DetachLight (Light* pkLight);
    void DetachAllLights ();

protected:
    std::vector<ObjectPtr> m_kLights;
};
```

The use of `ObjectPtr` instead of `LightPtr` resolves a circular header dependency between `Spatial` and `Light`.

Function `AttachLight` checks to see if the input light is already in the array. If so, no action is taken. If not, the light is added to the array. The function `GetLightQuantity` just returns the number of lights in the array. The function `GetLight` returns the *i*th light in the array. The function `DetachLight` searches the array for the input light, removing it if it exists.

In Wild Magic version 3, the `Geometry` class stored an array of `Light` objects. These lights were those occurring along a path from the root of the scene to the leaf geometry object. The lights were passed to the renderer so that lighting could be enabled and light parameters set, but this was all relative to having a fixed-function pipeline. Wild Magic version 4 is a shader-based engine, so now lighting is handled by shaders. Rather than requiring users to add lighting support to each and every shader written, the lights affecting the `Geometry` object are used by a local effect attached to the object, namely, a `LightingEffect` object. When such an effect exists, the `Renderer::Draw` function will rasterize the geometric primitive first using lighting, and then blend in any remaining effects attached to the object. This does amount to a multipass operation, so if you want only single-pass drawing, you must either stitch the lighting shaders to your own or provide shaders that can handle lighting in addition to whatever effects you had previously.

The render-state update, discussed later in this section, shows how the lights are propagated to the leaf nodes and combined into a lighting effect.

4.3.3 EFFECTS

In Wild Magic version 3, I added a new class called `Effect`, which stored vertex colors, textures, and texture coordinates. The object also had a `Renderer` function pointer for that function responsible for drawing an object to which the `Effect` was attached. One problem with this approach was that the vertex attributes (colors and texture coordinates) were stored separately from other vertex attributes for a geometric primitive. As much as I wanted to believe this allowed better sharing of data, it turned out to be a poor choice. The renderer had to look in multiple places to assemble all the data to render a single geometric primitive. Now the vertex buffers in Wild Magic version 4 encapsulate all vertex attributes for a geometric primitive.

Another problem prior to version 4 was that multipass special effects required you to create an `Effect`-derived class, create a corresponding function and add a pure virtual function to the `Renderer` interface, and then implement that function in each `Renderer`-derived class. The potential for frequent modifications to the core engine was quite large. In Wild Magic 4, the roles have been reversed. The `Effect` class is a simple abstract base class with a single virtual function:

```
virtual void Draw (Renderer* pkRenderer,
                  Spatial* pkGlobalObject, int iVisibleQuantity,
                  VisibleObject* akVisible);
```

Local effects that only need standard vertex attributes and textures never need this function. The class `ShaderEffect` is derived from `Effect` and is used to represent local effects. However, global effects such as projected shadows and planar reflections are represented by classes derived from `Effect`. These classes override the `Draw` function and make calls to the `Renderer` to achieve the desired effects. The `Renderer` class is powerful enough to supply the needs of any such global effect, so the `Renderer` code

does not need to change when you add new effects. Thus, the core engine is isolated from the addition of new effects. This is yet another manifestation of the open-closed principle of object-oriented programming. The core engine is closed to changes, thus protecting its integrity, but it is open to changes in that its interface supports the needs of any new features added to the special effects system.

The `Spatial` class has an interface to support attaching and detaching `Effect` objects. If the `Effect` is attached to a `Geometry` object, the effect is considered to be a local effect. Otherwise, the `Effect` is attached to a `Node` object and the effect is considered to be a global effect. Effects such as projected textures have both flavors. They can be attached as global effects, but they have no need to override the `Effect::Draw` function. The default `Draw` just iterates over the affected objects as shown:

```
void Effect::Draw (Renderer* pkRenderer,
    Spatial* pkGlobalObject, int iVisibleQuantity,
    VisibleObject* akVisible)
{
    // The default drawing function for global effects.
    // Essentially, this is a local effect applied to
    // all the visible leaf geometry.
    VisibleObject* pkCurrent = akVisible;
    for (int i = 0; i < iVisibleQuantity; i++, pkCurrent++)
    {
        Geometry* pkGeometry = (Geometry*)pkCurrent->Object;
        pkGeometry->AttachEffect(this);
        pkRenderer->Draw(pkGeometry);
        pkGeometry->DetachEffect(this);
    }
}
```

The inputs to this function are automatically handled by the renderer layer. Your application has no responsibility for making this happen.

The effects interface for `Spatial` is

```
class Spatial : public Object
{
public:
    int GetEffectQuantity () const;
    Effect* GetEffect (int i) const;
    void AttachEffect (Effect* pkEffect);
    void DetachEffect (Effect* pkEffect);
    void DetachAllEffects ();

protected:
    std::vector<EffectPtr> m_kEffects;
};
```

The interface is identical in structure to the one for attaching and detaching lights. One important note: Wild Magic version 3 allowed only a single `Effect` per `Spatial` object. This made it very difficult to obtain multipass drawing, requiring you to write your own renderer functions to support this. Wild Magic version 4 allows you to attach more than one `Effect`. The `Renderer::Draw` function automatically takes care of the multipass drawing—you are responsible only for specifying how the passes are blended together. See Section 3.4 for the details.

4.4 THE UPDATE PASS

This section describes the two types of updates for a scene graph. The geometric-state update must occur when you change vertex positional or normal data, transformation values, or the topology of the scene graph (attach or detach subtrees). The render-state update must occur when you attach or detach global render states, lights, or effects, or when the topology of the scene graph changes.

4.4.1 GEOMETRIC-STATE UPDATES

Recall that the scene graph management core classes are `Spatial`, `Geometry`, and `Node`. The `Spatial` class encapsulates the local and world transformations, the world bounding volume, and the parent pointer in support of the scene hierarchy. The `Geometry` class encapsulates the model data and the model bounding sphere and may exist only as leaf nodes in the scene hierarchy. The `Node` class encapsulates grouping and has a list of child pointers. All three classes participate in the *geometric update* of a scene hierarchy—the process of propagating transformations from parents to children (the downward pass), and then merging bounding volumes from children to parents (the upward pass).

The data members of the `Spatial` interface relevant to geometric updates are shown in the following partial interface listing:

```
class Spatial : public Object
{
public:
    Transformation Local;
    Transformation World;
    bool WorldIsCurrent;

    BoundingBoxPtr WorldBound;
    bool WorldBoundIsCurrent;
};
```

The data members are in public scope. This is a deviation from my choices for Wild Magic version 2, where the data members were protected or private and exposed only through public accessor functions, most of them implemented as inline functions. My choice for later versions was to reduce the verbosity, so to speak, of the class interface. In earlier versions, you would have a protected or private data member, one or more accessors, and inline implementations of those accessors. For example,

```
// in OldSpatial.h
class OldSpatial : public Object
{
public:
    Transformation& Local ();                // read-write access
    const Transformation& GetLocal () const; // read-only access
    void SetLocal (const Transformation& rkLocal); // write-only access
protected:
    Transformation m_kLocal;
};

// in OldSpatial.inl
Transformation& OldSpatial::Local ()
{ return m_kLocal; }
const Transformation& OldSpatial::GetLocal () const
{ return m_kLocal; }
void OldSpatial::SetLocal (const Transformation& rkLocal)
{ m_kLocal = rkLocal; }
```

The object-oriented premise of such an interface is to allow the underlying implementation of the class to change without forcing clients of the class to have to change their code. This is an example of *modular continuity*; see [Mey88, Section 2.1.4], specifically the following paragraph:

A design method satisfies Modular Continuity if a small change in a problem specification results in a change of just one module, or few modules, in the system obtained from the specification through the method. Such changes should not affect the *architecture* of the system, that is to say the relations between modules.

The interface for `OldSpatial` is a conservative way to achieve modular continuity. The experiences of two versions of Wild Magic led me to conclude that exposing some data members in the public interface is acceptable as long as the subsystem involving those data members is stable; that is, the subsystem will not change as the engine evolves. This is a less conservative way to achieve modular continuity because it relies on you not to change the subsystem.

By exposing data members in the public interface, you have another issue of concern. The function interfaces to data members, as shown in `OldSpatial`, can hide

side effects. For example, the function `SetLocal` has the responsibility of setting the `m_kLocal` data member of the class. But it could also perform operations on other data members or call other member functions, thus causing changes to state elsewhere in the system. If set/get function calls require side effects, it is not recommended that you expose the data member in the public interface. For if you were to do so, the engine user would have the responsibility for doing whatever is necessary to make those side effects occur.

In the case of the `Spatial` class, the `Local` data member is in public scope. Setting or getting the value has no side effects. The new interface is

```
// in Spatial.h
class Spatial : public Object
{
public:
    Transformation Local; // read-write access
};
```

and is clearly much reduced from that of `OldSpatial`. Observe that the prefix convention for variables is now used only for protected or private members. The convention for public data members is not to use prefixes and to capitalize the first letter of the name, just like function names are handled.

In class `Spatial` the world transformation is also in public scope. Recalling the previous discussion about transformations, the world transformations are compositions of local transformations. In this sense, a world transformation is computed as a (deferred) side effect of setting local transformations. I just mentioned that exposing data members in the public interface is not a good idea when side effects must occur, so why already violate that design goal? The problem has to do with the complexity of the controller system. Some controllers might naturally be constructed to *directly set the world transformations*. Indeed, the engine has a skin-and-bones controller that computes the world transformation for a triangle mesh. In a sense, the controller bypasses the standard mechanism that computes world transformations from local ones. The data members `World` and `WorldIsCurrent` are intended for read access by application writers but may be used for write access by controllers. If a controller sets the `World` member directly, it should also set the `WorldIsCurrent` flag to let the geometric update system know that the world transformation for this node should not be computed as a composition of its parent's world transformation and its local transformation.

Similar arguments apply to the data members `WorldBound` and `WorldBoundIsCurrent`. In some situations you have a node (and subtree) whose behavior is known to you (by design) and whose world bounding volume may be assigned directly. For example, the node might be a room in a building that never moves. The child nodes correspond to objects in the room; those objects can move within the room, so their world bounding volumes change. However, the room's world bounding volume

need not change. You may set the room's world bounding volume, but the geometric update system should be told not to recalculate that bounding volume from the child bounding volumes. The flag `WorldBoundIsCurrent` should be set to true in this situation.

The member functions of `Spatial` relevant to geometric updates are shown in the following partial interface listing:

```
class Spatial : public Object
{
public:
    void UpdateGS (double dAppTime = -Mathd::MAX_REAL,
                  bool bInitiator = true);
    void UpdateBS ();

protected:
    virtual void UpdateWorldData (double dAppTime);
    virtual void UpdateWorldBound () = 0;
    void PropagateBoundToRoot ();
};
```

The public functions `UpdateGS` (update geometric state) and `UpdateBS` (update bound state) are the entry points to the geometric update system. The function `UpdateGS` is for both propagation of transformations from parents to children and propagation of world bounding volumes from children to parents. The `dAppTime` (application time) is passed so that any animated quantities needing the current time to update their state have access to it. The Boolean parameter will be explained later. The function `UpdateBS` is for propagation only of world bounding volumes. The protected function `UpdateWorldData` supports the propagation of transformations in the downward pass. It is virtual to allow derived classes to update any additional world data that is affected by the change in world transformations. The protected functions `UpdateWorldBound` and `PropagateToRoot` support the calculation of world bounding volumes in the upward pass. The `UpdateWorldBound` function is pure virtual to require `Geometry` and `Node` to implement it as needed.

The portion of the `Geometry` interface relevant to geometric updates is

```
class Geometry : public Spatial
{
public:
    BoundingBoxPtr ModelBound;
    VertexBufferPtr VBuffer;
    IndexBufferPtr IBuffer;

    void UpdateMS (bool bUpdateNormals = true);
```

```
protected:
    virtual void UpdateModelBound ();
    virtual void UpdateModelNormals ();
    virtual void UpdateWorldBound ();
};
```

As with the `Spatial` class, the data members are in public scope because there are no immediate side effects from reading or writing them. But there are side effects that the programmer must ensure, namely, the geometric update itself.

The function `UpdateMS` (update model state) is the entry point into the update of the model bound and model normals. The function should be called whenever you change the model vertices. All that `UpdateMS` does is call the protected functions `UpdateModelBound` and `UpdateModelNormals`. The function `UpdateModelBound` computes a model bounding volume from the collection of vertices. This is accomplished by a call to the `BoundingBoxVolume` function `ComputeFromData`. I made the model bound update a virtual function just in case a derived class needs to compute the bound differently. For example, a derived class might have prior knowledge about the model bound and not even have to process the vertices.

The function `UpdateModelNormals` has an empty body in `Geometry` since the geometry class is just a container for vertices and normals. Derived classes need to implement `UpdateModelNormals` for their specific data representations. Not all derived classes have normals (for example, `PolyPoint` and `PolyLine`), so I decided to let them use the empty base class function rather than making the base function pure virtual and then requiring derived classes to implement it with empty functions.

The function `UpdateWorldBound` is an implementation of the pure virtual function in `Spatial`. All that it does is compute the world bounding volume from the model bounding volume by applying the current world transformation.

The member functions of `Node` relevant to geometric updates are shown in the following partial interface listing:

```
class Node : public Spatial
{
protected:
    virtual void UpdateWorldData (double dAppTime);
    virtual void UpdateWorldBound ();
};
```

The function `UpdateWorldData` is an implementation of the virtual function in the `Spatial` base class. It has the responsibility to propagate the geometric update to its children. The function `UpdateWorldBound` is an implementation of the pure virtual function in the `Spatial` base class. Whereas the `Geometry` class implements this to calculate a single world bounding volume for its data, the `Node` class implements this to compute a world bounding volume that contains the world bounding volume of all its children.

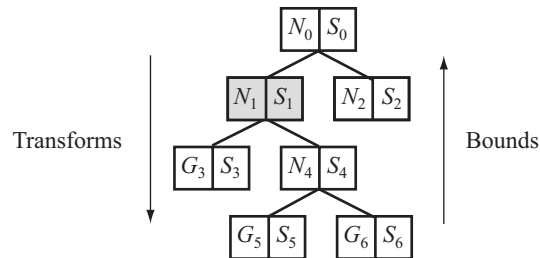


Figure 4.6 A geometric update of a simple scene graph. The shaded gray node, N_1 , is the one at which the `UpdateGS` call initiates.

Figure 4.6 illustrates the behavior of the update. The symbols are N for Node, S for Spatial, and G for Geometry. The rectangular boxes represent the nodes in the scene hierarchy. The occurrence of both an N and an S at a node stresses the fact that the Node is derived from Spatial, so both classes' public and protected interfaces are available to Node. A similar statement is made for Geometry and Spatial.

If the model bounding volumes or the model normals for a Geometry object are not current, that object must call `Geometry::UpdateMS()` to make them current. In most cases, the model data is current, for example, in rigid triangle meshes; you will not call `UpdateMS` often for such objects. The other extreme is something like a morph controller that changes the vertex data frequently, and the `UpdateMS` call occurs after each change.

Assuming the model data is current at all leaf nodes, the shaded gray box in the figure indicates that node N_1 is the one initiating a geometric update because its local transformation was changed (translation, rotation, and/or uniform scale). Its world transformation must be recomputed from its parent's (N_0) world transformation and its newly changed local transformation. The new world transformation is passed to its two children, G_3 and N_4 , so that they also may recompute their world transformations. The world bounding volume for G_3 must be recomputed from its model bounding volume. The process is repeated at node N_4 . Its world transformation is recomputed from the world transformation of N_1 and its local transformation. The new world transformation is passed to its two children, G_5 and G_6 , so that they may recompute their world transformations. Those leaf nodes also recompute their world bounding volumes from their new world transformations and their current model bounding volumes. On return to the parent N_4 , that node must recompute its world bounding volume to contain the new world bounding volumes of its children. On return to the node N_1 , that node must recompute its world bounding volume to contain the new world bounding volumes for G_3 and N_4 . You might think the geometric update terminates at this time, but not yet. The change in world bounding volume at N_1 can cause the world bounding volume of its parent, N_0 , to be out

of date. N_0 must be told to update itself. Generally, the change in world bounding volume at the initiator of the update must propagate all the way to the root of the scene hierarchy. Now the geometric update is complete. The sequence of operations is listed as pseudocode in the following pseudocode block. The indentation denotes the level of the recursive call of `UpdateGS`.

```
double dAppTime = <current application time>;
N1.UpdateGS(appTime,true);
    N1.World = compose(N0.World,N1.Local);
    G3.UpdateGS(appTime,false);
        G3.World = Compose(N1.World,G3.Local);
        G3.WorldBound = Transform(G3.World,G3.ModelBound);
    N4.UpdateGS(appTime,false);
        N4.World = Compose(N1.World,N4.Local);
        G5.UpdateGS(appTime,false);
            G5.World = Compose(N4.World,G5.Local);
            G5.WorldBound = Transform(G5.World,G5.ModelBound);
        G6.UpdateGS(appTime,false);
            G6.World = Compose(N4.World,G6.Local);
            G6.WorldBound = Transform(G6.World,G6.ModelBound);
        N4.WorldBound = BoundContaining(G5.WorldBound,G6.WorldBound);
    N1.WorldBound = BoundContaining(G3.WorldBound,N4.WorldBound);
    N0.WorldBound = BoundContaining(N1.WorldBound,N2.WorldBound);
```

The Boolean parameter `bInitiator` in the function `UpdateGS` is quite important. In the example, the `UpdateGS` call initiated at N_1 . A depth-first traversal of the subtree rooted at N_4 is performed, and the transformations are propagated downward. Once you reach a leaf node, the new world bounding volume is propagated upward. When the last child of N_1 has been visited, we found we needed to propagate its world bounding volume to its predecessors all the way to the root of the scene, in the example to N_0 . The propagation of a world bounding volume from G_5 to N_4 is slightly different than the propagation of a world bounding volume from N_1 to N_0 . The depth-first traversal at N_1 guarantees that the world bounding volumes are processed on the upward return. You certainly would not want each node to propagate its world bounding volume all the way to the root whenever that node is visited in the traversal, because only the initiator has that responsibility. If you were to have missed that subtlety and not had a Boolean parameter, the previous pseudocode would become

```
double dAppTime = <current application time>;
N1.UpdateGS(appTime);
    N1.World = compose(N0.World,N1.Local);
    G3.UpdateGS(appTime);
```



```

G3.World = Compose(N1.World,G3.Local);
G3.WorldBound = Transform(G3.World,G3.ModelBound);
N1.WorldBound = BoundContaining(G3.WorldBound,N4.WorldBound);
N0.WorldBound = BoundContaining(N1.WorldBound,N2.WorldBound);
N4.UpdateGS(appTime);
N4.World = Compose(N1.World,N4.Local);
G5.UpdateGS(appTime);
G5.World = Compose(N4.World,G5.Local);
G5.WorldBound = Transform(G5.World,G5.ModelBound);
N4.WorldBound = BoundContaining(G5.WorldBound,G6.WorldBound);
N1.WorldBound = BoundContaining(G3.WorldBound,N4.WorldBound);
N0.WorldBound = BoundContaining(N1.WorldBound,N2.WorldBound);
G6.UpdateGS(appTime);
G6.World = Compose(N4.World,G6.Local);
G6.WorldBound = Transform(G6.World,G6.ModelBound);
N4.WorldBound = BoundContaining(G5.WorldBound,G6.WorldBound);
N1.WorldBound = BoundContaining(G3.WorldBound,N4.WorldBound);
N0.WorldBound = BoundContaining(N1.WorldBound,N2.WorldBound);
N4.WorldBound = BoundContaining(G5.WorldBound,G6.WorldBound);
N1.WorldBound = BoundContaining(G3.WorldBound,N4.WorldBound);
N0.WorldBound = BoundContaining(N1.WorldBound,N2.WorldBound);
N1.WorldBound = BoundContaining(G3.WorldBound,N4.WorldBound);
N0.WorldBound = BoundContaining(N1.WorldBound,N2.WorldBound);

```

Clearly, this is an inefficient chunk of code. The Boolean parameter is used to prevent subtree nodes from propagating the world bounding volumes to the root.

The actual update code is shown next because I want to make a few comments about it. The entry point for the geometric update is

```

void Spatial::UpdateGS (double dAppTime, bool bInitiator)
{
    UpdateWorldData(dAppTime);
    UpdateWorldBound();
    if (bInitiator)
    {
        PropagateBoundToRoot();
    }
}

```

If the object is a Node object, the function UpdateWorldData propagates the transformations in the downward pass. If the object is a Geometry object, the function is not implemented in that class, and the Spatial version is used. The two different functions are

```

void Node::UpdateWorldData (double dAppTime)
{
    Spatial::UpdateWorldData(dAppTime);

    for (int i = 0; i < (nit)m_kChild.size(); i++)
    {
        Spatial* pkChild = m_kChild[i];
        if (pkChild)
        {
            pkChild->UpdateGS(dAppTime,false);
        }
    }
}

void Spatial::UpdateWorldData (double dAppTime)
{
    UpdateControllers(dAppTime);

    // NOTE: Updates on controllers for global state and lights
    // go here. To be discussed later.

    if (!WorldIsCurrent)
    {
        if (m_pkParent)
        {
            Transformation::Product(m_pkParent->World,Local,World);
        }
        else
        {
            World = Local;
        }
    }
}

```

The `Spatial` version of the function has the responsibility for computing the composition of the parent's world transformation and the object's local transformation, producing the object's world transformation. At the root of the scene (`m_pkParent` is `NULL`), the local and world transformations are the same. If a controller is used to compute the world transformation, then the Boolean flag `WorldIsCurrent` is true and the composition block is skipped. The `Node` version of the function allows the base class to compute the world transformation, and then it propagates the call (recursively) to its children. Observe that the `bInitiator` flag is set to false for the

child calls to prevent them from propagating the world bounding volumes to the root node.

The controller updates might or might not affect the transformation system. For example, the point, particles, and morph controllers all modify the model-space vertices (and possibly the model-space normals). Each of these call `UpdateMS` to guarantee the model bounding volume is current. Fortunately, this step occurs before our `UpdateGS` gets to the stage of updating world bounding volumes. Keyframe and inverse kinematics controllers modify local transformations, but they do not set the `WorldIsCurrent` flag to true because the world transformations must still be updated. The skin controllers modify the world transformations directly and do set the `WorldIsCurrent` flag to true.

In `UpdateGS`, on return from `UpdateWorldData` the world bounding volume is updated by `UpdateWorldBound`. If the object is a `Node` object, a bound of bounds is computed. If the object is a `Geometry` object, the newly computed world transformation is used to transform the model bounding volume to the world bounding volume.

```
void Node::UpdateWorldBound ()
{
    if (!WorldBoundIsCurrent)
    {
        bool bFoundFirstBound = false;
        for (int i = 0; i < (int)m_kChild.size(); i++)
        {
            Spatial* pkChild = m_kChild[i];
            if (pkChild)
            {
                if (bFoundFirstBound)
                {
                    // Merge current world bound with child
                    // world bound.
                    WorldBound->GrowToContain(pkChild->WorldBound);
                }
                else
                {
                    // Set world bound to first nonnull child
                    // world bound.
                    bFoundFirstBound = true;
                    WorldBound->CopyFrom(pkChild->WorldBound);
                }
            }
        }
    }
}
```

```

void Geometry::UpdateWorldBound ()
{
    ModelBound->TransformBy(World,WorldBound);
}

```

If the application has explicitly set the world bounding volume for the node, it should have also set `WorldBoundIsCurrent` to `false`, in which case `Node::UpdateWorldBound` has no work to do. However, if the node must update its world bounding volume, it does so by processing its child bounding volumes one at a time. The bounding volume of the first (nonnull) child is copied. If a second (nonnull) child exists, the current world bounding volume is modified to contain itself and the bound of the child. The growing algorithm continues until all children have been visited.

For bounding spheres, the iterative growing algorithm amounts to computing the smallest volume of two spheres, the current one and that of the next child. This is a greedy algorithm and does not generally produce the smallest volume bounding sphere that contains all the child bounding spheres. The algorithm to compute the smallest volume sphere containing a set of spheres is a very complicated beast [FG03]. The computation time is not amenable to real-time graphics, so instead we use a less exact bound, but one that can be computed quickly.

EXERCISE 4.1

Generating a bounding sphere incrementally as described here can lead to a final bounding sphere that is much larger than need be. Design a different algorithm that builds the final bounding sphere by considering all the input spheres at one time but that does not attempt to produce the minimum-volume bounding sphere. ■

The last stage of `UpdateGS` is to propagate the world bounding volume from the initiator to the root. The function that does this is `PropagateBoundToRoot`. This, too, is a recursive function, just through a linear list of nodes:

```

void Spatial::PropagateBoundToRoot ()
{
    if (m_pkParent)
    {
        m_pkParent->UpdateWorldBound();
        m_pkParent->PropagateBoundToRoot();
    }
}

```

As mentioned previously, if a local transformation has not changed at a node, but some geometric operations cause the world bounding volume to change, there is no reason to waste time propagating transformations in a downward traversal of the tree. Instead, just call `UpdateBS` to propagate the world bounding volume to the root:

Table 4.1 Updates that must occur when geometric quantities change.

<i>Changing Quantity</i>	<i>Required Updates</i>	<i>Top-Level Function to Call</i>
Model data	Model bound, model normals (if any)	Geometry::UpdateMS
Model bound	World bound	Spatial::UpdateGS or Spatial::UpdateBS
World bound	Parent world bound (if any)	Spatial::UpdateGS or Spatial::UpdateBS
Local transformation	World transformation, child transformations	Spatial::UpdateGS
World transformation	World bound	Spatial::UpdateGS

```

void Spatial::UpdateBS ()
{
    UpdateWorldBound();
    PropagateBoundToRoot();
}

```

Table 4.1 is a summary of the updates that must occur when various geometric quantities change in the system. All of the updates may be viewed as side effects to changes in the geometric state of the system. None of the side effects occur automatically, because I want application writers to use as much of their knowledge as possible about their environments and not force an inefficient update mechanism to occur behind the scenes.

For example, Figure 4.7 shows a scene hierarchy that needs updating. The light gray shaded nodes in the scene have had their local transformations changed. You could blindly call

```

a.UpdateGS(appTime,true);
b.UpdateGS(appTime,true);
c.UpdateGS(appTime,true);
d.UpdateGS(appTime,true);

```

to perform the updates, but this is not efficient. All that is needed is

```

a.UpdateGS(appTime,true);
b.UpdateGS(appTime,true);

```

Nodes c and d are updated as a side effect of the update at node a. In general, the minimum number of UpdateGS calls needed is the number of nodes requiring an update

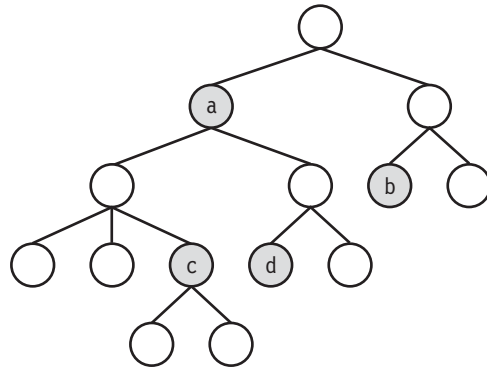


Figure 4.7 A scene hierarchy that needs updating. The shaded gray nodes have had their local transformations changed.

that have no predecessors that also require an update. Node a requires an update but has no out-of-date predecessors. Node c requires an update, but it has a predecessor, node a, that does. Although it is possible to construct an automated system to determine the minimum number of UpdateGS calls, that system will consume too many cycles. I believe it is better to let the application writers take advantage of knowledge they have about what is out of date and specifically call UpdateGS themselves.

4.4.2 RENDER-STATE UPDATES

The core classes Spatial, Geometry, and Node all have some form of support for storing render state and making sure that the renderer has the complete state for each object it draws. The class Geometry has the storage capabilities for the render state that affects it. My decision to do this in Wild Magic version 3 was to provide a single object type (Geometry) to the renderer. Wild Magic version 2 had an abstract rendering that required the object to be passed as the specific types they were, but the interface was cumbersome. The redesign for version 3 made the rendering interface much more streamlined. The process of assembling the information in the Geometry object is referred to as *updating the render state*.

The portions of the interfaces for classes Spatial, Node, and Geometry that are relevant to updating the render state are

```
class Spatial : public Object
{
public:
    virtual void UpdateRS (
```

```

        std::vector<GlobalState*>* akGStack = 0,
        std::vector<Light*>* pkLStack = 0);

protected:
    void PropagateStateFromRoot (
        std::vector<GlobalState*>* akGStack,
        std::vector<Light*>* pkLStack);

    void PushState (
        std::vector<GlobalState*>* akGStack,
        std::vector<Light*>* pkLStack);

    void PopState (
        std::vector<GlobalState*>* akGStack,
        std::vector<Light*>* pkLStack);

    virtual void UpdateState (
        std::vector<GlobalState*>* akGStack,
        std::vector<Light*>* pkLStack) = 0;
};

class Node : public Object
{
protected:
    virtual void UpdateState (
        std::vector<GlobalState*>* akGStack,
        std::vector<Light*>* pkLStack);
};

class Geometry : public Object
{
protected:
    virtual void UpdateState (
        std::vector<GlobalState*>* akGStack,
        std::vector<Light*>* pkLStack);
};

```

The entry point into the system is method `UpdateRS` (update render state). The input parameters are containers to assemble the global state and lights during a depth-first traversal of the scene hierarchy. The parameters have default values. The caller of `UpdateRS` should not set these but should just call `object.UpdateRS()`. The containers are allocated and managed internally by the update system. The containers are treated as if they were stacks.

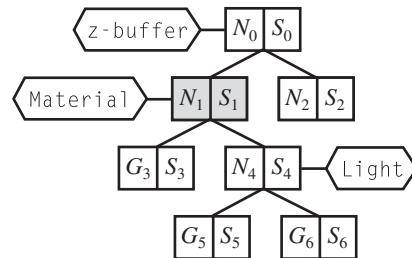


Figure 4.8 A common situation for updating render state.

The protected member functions are helper functions for the depth-first traversal. The function `PushState` pushes any global state and lights that the `Spatial` object has attached to it onto stacks. The function `PopState` pops those stacks. The intent is that the stacks are used by all the nodes in the scene hierarchy as they are visited. Function `Node::UpdateState` has the responsibility for propagating the update in a recursive traversal of the scene hierarchy. Function `Geometry::UpdateState` is called at leaf nodes of the hierarchy. It has the responsibility for copying the contents of the global-state stack into its appropriate data members, but it assembles the lights into a `LightEffect` object to be used by the renderer as the first pass of a multipass drawing operation. The stacks store smart pointers to global states and lights, so the copy is really a smart pointer copy and the objects are shared.

The render state at a leaf node represents all the global states and lights that occur on the path from the root node to the leaf node. However, the `UpdateRS` call need only be called at a node whose subtree needs a render-state update. Figure 4.8 illustrates a common situation.

The `z-buffer` state is already attached to node N_0 , and the light is already attached to node N_4 . A material state is attached to node N_1 . The render-state update is initiated at N_1 . The result of the depth-first traversal of the subtree at N_1 is the following: G_3 has links to the `z-buffer` and material states; G_5 has links to the `z-buffer` state, the material state, and the light; and G_6 has links to the `z-buffer` state, the material state, and the light. The `z-buffer` state is, however, not in the subtree of N_1 , so we in fact have to start collecting the states from the root node and along paths that lead to the leaf nodes that are in the subtree of N_1 . The function `PropagateStateFromRoot` has the responsibility of starting the render state update at N_1 by first traversing to the root N_0 , collecting the render state of the path from N_0 to N_1 , and then passing this state to the leaf nodes of the subtree at N_1 , together with any additional render state that is in that subtree. Pseudocode for the sequence of operations is listed next. The indentation denotes the level of the calling stack.


```

N1.UpdateRS();
N1: create global state stack GS;      // GS = {}
N1: create light stack LS;             // LS = {}
N1.PropagateStateFromRoot(GS,LS);
    N0.PropagateStateFromRoot(GS,LS);
        N0.PushState(GS,LS);           // GS = {zbuffer}, LS = {}
        N1.PushState(GS,LS);           // GS = {zbuffer,material},
                                         // LS = {}
N1.UpdateState(GS,LS);
    G3.UpdateRS(GS,LS);
        G3.PushState(GS,LS);           // GS = {zbuffer,material},
                                         // LS = {}
        G3.UpdateState(GS,LS);         // share: zbuffer,material
        G3.PopState(GS,LS);
    N4.UpdateRS(GS,LS);
        N4.PushState(GS,LS);           // GS = {zbuffer,material},
                                         // LS = {light}
    N4.UpdateState(GS,LS);
        G5.UpdateRS(GS,LS);
            G5.PushState(GS,LS);        // GS = {zbuffer,material},
                                         // LS = {light}
            G5.UpdateStore(GS,LS);      // share: zbuffer,material,
                                         // light
            G5.PopState(GS,LS);         // GS = {zbuffer,material},
                                         // LS = {light}
        G6.UpdateRS(GS,LS);
            G6.PushState(GS,LS);        // GS = {zbuffer,material},
                                         // LS = {light}
            G6.UpdateStore(GS,LS);      // share: zbuffer,material,
                                         // light
            G6.PopState(GS,LS);         // GS = {zbuffer,material},
                                         // LS = {light}
        N4.PopState(GS,LS);            // GS = {zbuffer,material},
                                         // LS = {}
    N1: destroy global state stack GS;  // GS = {}
    N1: destroy light stack LS;         // LS = {}

```

The pseudocode indicates the global-state stack is initially empty. In Wild Magic version 3, the stacks were actually not empty but initialized with the default render states. This is not necessary. In Wild Magic version 4, the stacks are initially empty. When the renderer finally receives a stack of render states, it uses whatever does exist to override the default states. Thus, the behavior is identical to that of Wild Magic version 3, but without having to push and pop the default states.

Nothing prevents you from having multiple states of the same type in a single path from root node to leaf node. For example, the root node can have a z-buffer state that enables depth buffering, but a subtree of objects at node N that can be correctly drawn without depth buffering enabled can also have a z-buffer state that disables depth buffering.

At first glance you might be tempted not to have `PropagateStateFromRoot` in the update system. Consider the current example. Before the material state was attached to node N_1 , and assuming the scene hierarchy was current regarding render state, G_3 should have in its local storage the z-buffer. G_5 and G_6 should each have local storage containing the z-buffer and light. When you attach the material to node N_1 and call `UpdateRS`, whose implementation does only the depth-first traversal, it appears that the correct states will occur at the geometry leaf nodes. In my implementation this is not the case. The global-state stacks are empty initially. The z-buffer stack will be empty when the traversal reaches the leaf nodes. The current z-buffer pointer in the `Geometry::UpdateState` global-state array is overwritten with a null pointer, in which case the z-buffer state of N_0 will be ignored, thus changing the behavior at the leaf nodes.

Now you might consider changing the render-state update semantics so that the global-state stack is initially empty, accumulate only the render states visited in the depth-first traversal, and then have `Geometry::UpdateState` copy only those pointers into its local storage. To throw a wrench into the works, suppose that the subtree at N_4 is detached from the scene and a new subtree added as the second child of N_1 . The leaf nodes of the new subtree are unaware of the render state that N_1 and its predecessors have. A call to the depth-first-only `UpdateRS` at N_1 will propagate the render states from N_1 downward, but now the z-buffer state of N_0 is missing from the leaf nodes. To remedy this problem, you should have called `UpdateRS` at the root node N_0 . The leaf nodes will get all the render state they deserve, but unfortunately other subtrees of the scene hierarchy are updated even though they have current render-state information. My decision to include `PropagateStateFromRoot` is based on having as efficient a render-state update as possible. In a situation such as the current example, the application writer does not have to call `UpdateRS` at N_0 when all that has changed is a subtree modification at N_4 . In my update system, after the subtree is replaced by a new one, you only need to call `UpdateRS` at N_4 .

The previous discussion does point out that there are various circumstances when you have to call `UpdateRS`. Clearly, if you attach a new global state or light to a node, you should call `UpdateRS` to propagate that information to the leaf nodes. Similarly, if you detach a global state or light from a node, the leaf nodes still have smart pointers to those. You must call `UpdateRS` to eliminate those smart pointers, replacing the global-state pointers with ones to the default global states. The light pointers are just removed from the storage. A change in the topology of the scene, such as attaching new children or replacing children at a node N , also requires you to call `UpdateRS`. This is the only way to inform the leaf nodes of the new subtree about their render state.

If you change the data members in a global-state object or in a light object, you do *not* have to call `UpdateRS`. The local storage of smart pointers in `Geometry` to the global states and lights guarantees that you are sharing those objects. The changes to the data members are immediately known to the `Geometry` object, so when the renderer goes to draw the object, it has access to the new values of the data members.

To finish up, here is a brief discussion of the implementations of the render-state update functions. The entry point is

```
void Spatial::UpdateRS (std::vector<GlobalState*>* akGStack,
    std::vector<Light*>* pkLStack)
{
    bool bInitiator = (akGStack == 0);

    if (bInitiator)
    {
        // The order of preference is
        // (1) Default global states are used.
        // (2) Geometry can override them, but if global
        //     state FOOBAR has not been pushed to the
        //     Geometry leaf node, then the current FOOBAR
        //     remains in effect (rather than the default
        //     FOOBAR being used).
        // (3) Effects can override default or Geometry
        //     render states.
        akGStack = WM4_NEW std::vector<GlobalState*>[
            GlobalState::MAX_STATE_TYPE];
        for (int i = 0; i < GlobalState::MAX_STATE_TYPE; i++)
        {
            akGStack[i].push_back(0);
        }

        // Stack has no lights initially.
        pkLStack = WM4_NEW std::vector<Light*>;

        // Traverse to root and push states from root to this
        // node.
        PropagateStateFromRoot(akGStack, pkLStack);
    }
    else
    {
        // Push states at this node.
        PushState(akGStack, pkLStack);
    }
}
```

```

    // Propagate the new state to the subtree rooted here.
    UpdateState(akGStack,pkLStack);

    if (bInitiator)
    {
        WM4_DELETE[] akGStack;
        WM4_DELETE pkLStack;
    }
    else
    {
        // Pop states at this node.
        PopState(akGStack,pkLStack);
    }
}

```

The initiator of the update calls `UpdateRS()` with no parameters. The default parameters are null pointers. This lets the function determine that the initiator is the one who is responsible for allocating and deallocating the stacks. Notice that the global-state “stack” is really an array of stacks, one stack per global-state type. The initiator is also responsible for calling `PropagateStateFromRoot`. The `UpdateState` call propagates the update to child nodes for a `Node` object but copies the smart pointers in the stacks to local storage for a `Geometry` object. For the noninitiators, the sequence of calls is effectively

```

PushState(akGStack,pkLStack);
UpdateState(akGStack,pkLStack);
PopState(akGStack,pkLStack);

```

In words: push my state onto the stacks, propagate it to my children, and then pop my state from the stacks.

The propagation of state from the root is

```

void Spatial::PropagateStateFromRoot (
    std::vector<GlobalState*>* akGStack,
    std::vector<Light*>* pkLStack)
{
    // Traverse to root to allow downward state propagation.
    if (m_pkParent)
    {
        m_pkParent->PropagateStateFromRoot(akGStack,pkLStack);
    }

    // Push states onto current render-state stack.
    PushState(akGStack,pkLStack);
}

```

This is a recursive call that traverses a linear list of nodes. The traversal takes you up the tree to the root, and then you push the states of the nodes as you return to the initiator.

The pushing and popping of state is straightforward:

```
void Spatial::PushState (
    std::vector<GlobalState*>* akGStack,
    std::vector<Light*>* pkLStack)
{
    int i;
    for (i = 0; i < (int)m_kGlobalStates.size(); i++)
    {
        int eType = m_kGlobalStates[i]->GetStateType();
        akGStack[eType].push_back(m_kGlobalStates[i]);
    }

    for (i = 0; i < (int)m_kLights.size(); i++)
    {
        Light* pkLight = StaticCast<Light>(m_kLights[i]);
        pkLStack->push_back(pkLight);
    }
}

void Spatial::PopState (
    std::vector<GlobalState*>* akGStack,
    std::vector<Light*>* pkLStack)
{
    int i;
    for (i = 0; i < (int)m_kGlobalStates.size(); i++)
    {
        int eType = m_kGlobalStates[i]->GetStateType();
        akGStack[eType].pop_back();
    }

    for (i = 0; i < (int)m_kLights.size(); i++)
    {
        pkLStack->pop_back();
    }
}
```

The code iterates over an array of global states attached to the object and pushes them on the stack (pops them from the stack) corresponding to the type of the state. The code also iterates over an array of lights attached to the object and pushes them on the stack (pops them from the stack).

The propagation of the update down the tree is

```
void Node::UpdateState (
    std::vector<GlobalState*>* akGStack,
    std::vector<Light*>* pkLStack)
{
    for (int i = 0; i < (int)m_kChild.size(); i++)
    {
        Spatial* pkChild = m_kChild[i];
        if (pkChild)
        {
            pkChild->UpdateRS(akGStack,pkLStack);
        }
    }
}
```

This, too, is a straightforward operation. Just as with the geometric update functions `UpdateGS` and `UpdateWorldData`, the pair `UpdateRS` and `UpdateState` form a recursive chain (A calls B, B calls A, etc.).

Finally, the copy of smart pointers from the stacks to local storage is

```
void Geometry::UpdateState (
    std::vector<GlobalState*>* akGStack,
    std::vector<Light*>* pkLStack)
{
    // Update global state.
    int i;
    for (i = 0; i < GlobalState::MAX_STATE_TYPE; i++)
    {
        GlobalState* pkGState = 0;
        pkGState = akGStack[i].back();
        States[i] = pkGState;
    }

    // Update lights.
    int iLQuantity = (int)pkLStack->size();
    if (iLQuantity > 0)
    {
        if (LEffect)
        {
            LEffect->DetachAllLights();
        }
        else
    }
```

```

    {
        LEffect = WM4_NEW LightingEffect;
    }

    for (i = 0; i < iLQuantity; i++)
    {
        LEffect->AttachLight((*pkLStack)[i]);
    }

    LEffect->Configure();
}
else
{
    LEffect = 0;
}
}

```

No surprises here, either. The Geometry class has an array of smart pointers to Global-State for global-state storage. In Wild Magic version 3, a list of lights was maintained to be passed to the renderer for setting up the dynamic lighting in the fixed-function pipeline. In Wild Magic version 4, a LightingEffect object is maintained to be passed to the renderer for the shader pipeline. This effect is applied first to the geometric primitive before any other attached effects.

4.5 THE CULLING PASS

Starting with version 4, the culling of objects in scene graphs is a new system in Wild Magic. In previous versions, the culling and drawing were done on the same pass: The scene tree was traversed. Entire subtrees were culled when their bounding volumes were outside the frustum. If the traversal eventually reached a potentially visible Geometry object at a leaf node, that object was drawn immediately in the Wild Magic version 2 engine. This was also the case for the Wild Magic version 3 engine unless you activated the deferred drawing scene.

Culling in Wild Magic 4 is done in a pass separate from drawing. Three new classes have been introduced: Culler, VisibleObject, and VisibleSet. The class VisibleObject is a simple structure:

```

class VisibleObject
{
public:
    Spatial* Object;
    Effect* GlobalEffect;
};

```

And the class `VisibleSet` is a container for objects of type `VisibleObject`:

```
class VisibleSet
{
public:
    VisibleSet (int iMaxQuantity = 0, int iGrowBy = 0);
    virtual ~VisibleSet ();

    int GetQuantity () const;
    VisibleObject* GetVisible ();
    VisibleObject& GetVisible (int i);

    void Insert (Spatial* pkObject,
                Effect* pkGlobalEffect);
    void Clear ();
    void Resize (int iMaxQuantity, int iGrowBy);

private:
    enum
    {
        VS_DEFAULT_MAX_QUANTITY = 32,
        VS_DEFAULT_GROWBY = 32,
    };

    int m_iMaxQuantity, m_iGrowBy, m_iQuantity;
    VisibleObject* m_akVisible;
};
```

As a scene graph is traversed for the purpose of culling, each potentially visible Geometry object is inserted into a `VisibleSet` object. By potentially visible, I mean that the bounding volume of the object overlaps the view frustum. It is possible that the object itself is not visible, but all we can conclude from the overlap is that the object might be visible. The corresponding `VisibleObject` has its `Spatial` pointer set to the Geometry pointer and the `Effect` pointer is set to null. However, if a potentially visible Node object with a global effect attached is encountered, the corresponding `VisibleObject` has its `Spatial` pointer set to the Node pointer and the `Effect` pointer is set to that of the global effect. This type of `VisibleObject` is a sentinel that marks the beginning of the scope of the global effect. All potentially visible objects in the subtree rooted at the Node are influenced by the global effect. Once the culling traversal returns from processing the subtree, another sentinel is added to the visible set, marking the end of the scope of the global effect. This system was described in Section 3.4.1.

The `Culler` class is designed to manage the culling of a scene and storing of the potentially visible objects in a `VisibleSet`. If you have looked at Wild Magic version 3, much of the work for culling was done by the `Camera` class. That subsystem has been factored out into the `Culler` class, which has quite a large interface. I will describe it in small pieces.

The constructor, destructor, and a couple of data members and accessors are

```
class Culler
{
public:
    Culler (int iMaxQuantity = 0, int iGrowBy = 0,
           const Camera* pkCamera = 0);
    virtual ~Culler ();

    void SetCamera (const Camera* pkCamera);
    const Camera* GetCamera () const;
    void SetFrustum (const float* afFrustum);
    const float* GetFrustum () const;
    virtual void Insert (Spatial* pkObject,
                       Effect* pkGlobalEffect);

protected:
    const Camera* m_pkCamera;
    float m_afFrustum[Camera::VF_QUANTITY];
};
```

The first two input parameters are used to create the set of potentially visible objects. The third parameter is a camera from whose perspective the objects are potentially visible. Although normally this is a camera relative to the observer, it can also be a projector for a light source for which you want to know about objects that will cast shadows. This parameter is assigned to the data member `m_pkCamera`. If the camera is not passed to the constructor, you should set it using `SetCamera` before attempting a culling operation; see the discussion in Section 4.5.1 about `ComputeVisibleSet`. The camera information might be needed during the culling pass, hence the storage of a pointer to it. A copy of the view frustum for the input camera is maintained. This allows various subsystems to change the frustum parameters during culling (e.g., the portal system) without affecting the camera, whose initial state is needed by the renderer.

The `Culler` maintains culling planes in world coordinates. These planes include the six view frustum planes. Additional planes may be set by the user, similar to user-defined clipping planes but for the purpose of culling. These are pushed and popped as needed. It is not possible to pop the view frustum planes—they will always be used for culling.

```

class Culler
{
    enum { VS_MAX_PLANE_QUANTITY = 32 };
    int GetPlaneQuantity () const;
    const Plane3f* GetPlanes () const;
    void SetPlaneState (unsigned int uiPlaneState);
    unsigned int GetPlaneState () const;
    void PushPlane (const Plane3f& rkPlane);
    void PopPlane ();

    bool IsVisible (const BoundingBox* pkWound);
    bool IsVisible (int iVertexQuantity, const Vector3f* akVertex,
        bool bIgnoreNearPlane);
    int WhichSide (const Plane3f& rkPlane) const;

protected:
    int m_iPlaneQuantity;
    Plane3f m_akPlane[VS_MAX_PLANE_QUANTITY];
    unsigned int m_uiPlaneState;
};

```

The member `m_uiPlaneState` represents bit flags to store whether or not a plane is active in the culling system. A bit of 1 means the plane is active; otherwise the plane is inactive. An active plane is compared to bounding volumes, whereas an inactive plane is not. This supports an efficient culling of a hierarchy. For example, if a node's bounding volume is inside the left plane of the view frustum, then the left plane is set to inactive because the children of the node are automatically inside the left plane.

The member function `IsVisible (const BoundingBox*)` compares the object's world bounding volume to the culling planes. This function is called only by `Spatial` during a traversal of the scene, which is discussed later in this section. The other member function `IsVisible` is used by the portal system in `Portal::GetVisibleSet`; see Section 6.3 for details.

The member function `WhichSide` is used in `BspNode::GetVisibleSet`. It determines if the view frustum is fully on one side of a plane. The “positive side” of the plane is the half space to which the plane normal points. The “negative side” is the other half space. The function returns +1 if the view frustum is fully on the positive side of the plane, -1 if the view frustum is fully on the negative side of the plane, or 0 if the view frustum straddles the plane. The input plane is in world coordinates and the world camera coordinate system is used for the test.

The remainder of the `Culler` interface has to do with the potentially visible set itself.

```

class Culler
{
public:
    VisibleSet& GetVisibleSet ();
    void ComputeVisibleSet (Spatial* pkScene);

protected:
    VisibleSet m_kVisible;
};

```

The potentially visible set is the member `m_kVisible` and an accessor to it is `GetVisibleSet`. The main entry point into the culling of a scene graph is the function `ComputeVisibleSet`, discussed next.

4.5.1 HIERARCHICAL CULLING

The `Spatial` and `Culler` classes must interact during a culling pass. The relevant interface in the `Spatial` class is

```

class Spatial : public Object
{
public:
    enum CullingMode
    {
        CULL_DYNAMIC,
        CULL_ALWAYS,
        CULL_NEVER
    };

    CullingMode Culling;

    void OnGetVisibleSet (Culler& rkCuller, bool bNoCull);
    virtual void GetVisibleSet (Culler& rkCuller, bool bNoCull) = 0;
};

```

Wild Magic version 3 had a Boolean flag, `ForceCull`. By setting this to true, you forced the subtree rooted at the node to be culled. This is a useful mechanism when you have prior knowledge that the objects in the subtree are not visible. In particular, you could precompute visibility for a game level by partitioning the world into cells. Each cell maintains a list of potentially visible objects. When the camera enters that cell, you can force-cull all objects known not to be visible from that cell.

It turns out that a three-way flag is more useful. Wild Magic version 4 adds this via the enumeration `CullingMode`. The enumeration `CULL_ALWAYS` corresponds

to setting ForceCull to true in Wild Magic version 2, and CULL_DYNAMIC corresponds to ForceCull being false in that version. However, there are times when it is useful not to cull an object when it is *not visible to the camera*. For example, this is used in the SampleGraphics/PlanarReflections sample application. A skinned biped has reflections on two planes—the floor and a wall. If the skinned biped is no longer visible to the camera, its reflection might still be visible. To avoid the reflection instantaneously disappearing when the skinned biped moves just outside the view frustum, you can set the culling mode to CULL_NEVER. If a node is tagged this way, all objects in its subtree inherit the attribute. This is accomplished by passing a Boolean flag (set to true) to the recursive chain of culling operations listed in the interface for Spatial, namely, OnGetVisibleSet and GetVisibleSet.

The entry point to culling is via the Culler function

```
void Culler::ComputeVisibleSet (Spatial* pkScene)
{
    SetFrustum(m_pkCamera->GetFrustum());
    m_kVisible.Clear();
    pkScene->OnGetVisibleSet(*this,false);
}
```

A copy of the camera's frustum is made via SetFrustum. Moreover, the view frustum planes are computed in world coordinates. This requires querying the camera for its current world coordinate system (eye point, direction vector, up vector, and right vector). The current potentially visible set is cleared; that is, it contains no elements. The final function call tells the scene to traverse itself, using the Culler to perform the culling tests and to store the potentially visible objects in the scene.

The called Spatial function is

```
void Spatial::OnGetVisibleSet (Culler& rkCuller, bool bNoCull)
{
    if (Culling == CULL_ALWAYS)
    {
        return;
    }

    if (Culling == CULL_NEVER)
    {
        bNoCull = true;
    }

    unsigned int uiSavePlaneState = rkCuller.GetPlaneState();
    if (bNoCull || rkCuller.IsVisible(WorldBound))
```

```

    {
        GetVisibleSet(rkCuller,bNoCull);
    }
    rkCuller.SetPlaneState(uiSavePlaneState);
}

```

The logic is straightforward. If the object requests that you cull it, the function simply returns—no internal nodes or geometry objects are inserted into the potentially visible set. If the object wants never to be culled, the `bNoCull` flag is set to true and passed to the recursive call `GetVisibleSet`. This will guarantee that all leaf geometry and all nodes with global effects will be added to the potentially visible set.

The `uiSavePlaneState` bit flag stores the current state of which planes are active and inactive. As mentioned previously, if a node's bounding volume is inside the left plane of the frustum, then all its child objects are inside the left plane. The left plane may be disabled when traversing the node's children, thus saving cycles by avoiding the bounding volume versus plane tests. When the traversal returns from the node's children, the previous state of the active/inactive planes must be restored (via `SetPlaneState`).

If the `Spatial` object is also a `Geometry` object, the function `GetVisibleSet` is

```

void Geometry::GetVisibleSet (Culler& rkCuller, bool)
{
    rkCuller.Insert(this,0);
}

```

The object is inserted into the culler's potentially visible set. Since this is a leaf node, the `Effect` pointer is not needed, so `NULL` is passed. If the `Spatial` object is also a `Node` object (an interior node of the tree), the function `GetVisibleSet` is

```

void Node::GetVisibleSet (Culler& rkCuller, bool bNoCull)
{
    if (m_kEffects.size() > 0)
    {
        // This is a global effect. Place a 'begin' sentinel
        // in the visible set to indicate the effect is
        // active.
        rkCuller.Insert(this,m_kEffects[0]);
    }

    // All Geometry objects in the subtree are added to the
    // visible set. If a global effect is active, the
    // Geometry objects in the subtree will be drawn using it.
    for (int i = 0; i < (int)m_kChild.size(); i++)

```

```

    {
        Spatial* pkChild = m_kChild[i];
        if (pkChild)
        {
            pkChild->OnGetVisibleSet(rkCuller,bNoCull);
        }
    }

    if (m_kEffects.size() > 0)
    {
        // Place an 'end' sentinel in the visible set to
        // indicate that the global effect is inactive.
        rkCuller.Insert(0,0);
    }
}

```

As mentioned in Section 3.4.1, the sentinels are used to guide the renderer in making certain that the global effects are correctly applied to the Geometry bound by the sentinels.

Various Spatial-derived classes override `GetVisibleSet` in order to correctly process the object and child objects (if any). These include switch nodes (`SwitchNode`), discrete-level-of-detail nodes (`DlodNode`), continuous-level-of-detail meshes (`ClodMesh`) and terrains (`ClodTerrain`), particle systems (`Particles`), and the classes associated with the portal visibility system (`BspNode`, `ConvexRegion`, and `ConvexRegionManager`). The `Portal` class has a similarly named function, but it is not derived from `Spatial`. The function is used, though, for culling.

After `ComputeVisibleSet` is called on a scene, the nonempty potentially visible set may then be passed to `Renderer::DrawScene` for drawing.

4.5.2 SORTED CULLING

The culling described in the previous section involves a depth-first traversal of the scene graph. Because the culling pass is separate from the drawing pass, you can process the potentially visible set any way you like before sending it on to the renderer for drawing. In particular, you can sort based on render state. The only warning is that your sorter must pay attention to the sentinels inserted when global effects were encountered. Specifically, if a `Geometry` object is contained by a begin-end pair of sentinels, and if your sorter causes this object to be moved outside that pair, the result will be incorrect rendering.

Another way to sort is during the culling itself. The virtual function `GetVisibleSet` may be overridden to obtain any desired semantics you want. The case of interest in *Wild Magic* is the portal system. When a scene graph contains a subgraph corresponding to a group of rooms connected by portals, the culling and drawing passes

cease to use depth-first search for the traversal. The traversal path is based on the visibility graph implied by the room-portal graph. One of the performance issues in a portal system is overdraw of the contents of a room because multiple portals lead into the room and the camera/observer can see through both portals from a single location. To avoid this, a class derived from `Culler` may be used for the portal system. It accumulates the potentially visible objects during the traversal of the room-portal graph but maintains a set of unique objects; that is, an object is never added twice to the set.

4.6 THE DRAWING PASS

The input to a culling pass is a scene graph. The output is a potentially visible set of objects. This set is passed to the renderer for drawing. The common structure of code in the applications has the abstraction

```
// *** application initialization code ***

// creation of objects
create Culler m_kCuller;
create Camera m_spkCamera;
create scene m_spkScene;

// initial update of objects
m_spkScene->UpdateGS();
m_spkScene->UpdateRS();

// initial culling of scene
m_kCuller.SetCamera(m_spkCamera);
m_kCuller.ComputeVisibleSet(m_spkScene);

// *** in the idle loop ***
if (MoveCamera())
{
    m_kCuller.ComputeVisibleSet(m_spkScene);
}

if (MoveObject())
{
    m_spkScene->UpdateGS();
    m_kCuller.ComputeVisibleSet(m_spkScene);
}
```

```

m_pkRenderer->ClearBuffers();
if (m_pkRenderer->BeginScene())
{
    m_pkRenderer->DrawScene(m_kCuller.GetVisibleSet());
    m_pkRenderer->EndScene();
}
m_pkRenderer->DisplayBackBuffer();

```

In the initialization code, the culler, camera, and scene are created and initialized for first use. In the idle loop, anytime the camera has moved or the scene has moved (i.e., objects in it were transformed), the potentially visible set can change. After computing the set for the current camera and scene, the renderer is ready to draw the objects in the set. This occurs in the `DrawScene` call. The actual work done by the renderer was discussed in Section 3.4. The discussion in this section is about how you manipulate the `Geometry` and `Node` objects to obtain the desired special effects.

In the following material, all Cg shader program files are located in the directory

`GeometricTools/WildMagic4/Data/ShaderPrograms/Cg`

and the compiled shader programs are in the directory

`GeometricTools/WildMagic4/Data/Wmsp`

The compound extensions on the compiled programs make it clear which renderers they belong to. All the files are ASCII text, so you can browse them with your favorite text browser.

4.6.1 SINGLE-PASS DRAWING

The simplest and most efficient shader effects use single-pass drawing. The `ShaderEffect` class allows you to create an effect with a specified number of passes, in this case one. Alpha blending parameters are set by you only if you plan on blending the rendered results with the current contents of the color buffer (in the frame buffer or in an offscreen buffer).

An example of a single-pass effect is `TextureEffect`, a class derived from `ShaderEffect`. The constructor for this class calls the base-class constructor, setting the number of passes to one.

```

TextureEffect::TextureEffect (const std::string& rkBaseName)
:
    ShaderEffect(1)
{
    m_kVShader[0] = WM4_NEW VertexShader("Texture");
    m_kPShader[0] = WM4_NEW PixelShader("Texture");
}

```



```

    m_kPShader[0]->SetTextureQuantity(1);
    m_kPShader[0]->SetImageName(0,rkBaseName);
}

```

A vertex and a pixel shader are created, both based on the Cg code in the file `Texture.cg`.

A sample block of application code for creating a scene containing a single triangle is

```

// The root node of the scene. NOTE: An application can draw
// multiple scenes during a single iteration through the idle
// loop. It is not necessary that everything in the world be
// placed in one scene graph.
m_spkScene = WM4_NEW Node;

// Create a single triangle whose vertices have positions and
// 2D texture coordinates (in unit 0).
Attributes kAttr;
kAttr.SetPChannels(3);
kAttr.SetTChannels(0,2);
VertexBuffer* pkVBuffer = WM4_NEW VertexBuffer(kAttr,3);
pkVBuffer->Position3(0) = Vector3f(1.0f,0.0f,0.0f);
pkVBuffer->Position3(1) = Vector3f(0.0f,1.0f,0.0f);
pkVBuffer->Position3(2) = Vector3f(0.0f,0.0f,1.0f);
pkVBuffer->TCoord2(0,0) = Vector2f(0.0f,0.0f);
pkVBuffer->TCoord2(0,1) = Vector2f(1.0f,0.0f);
pkVBuffer->TCoord2(0,2) = Vector2f(0.0f,1.0f);
IndexBuffer* pkIBuffer = WM4_NEW IndexBuffer(3);
int* aiIndex = pkIBuffer->GetData();
aiIndex[0] = 0; aiIndex[1] = 1; aiIndex[2] = 2;
TriMesh* pkTriangle = WM4_NEW TriMesh(pkVBuffer,pkIBuffer);
m_spkScene->AttachChild(pkTriangle);

// Create a single-pass texture effect.
Effect* pkEffect = WM4_NEW TextureEffect("MyImage");

// Attach the effect to the geometric primitive.
pkTriangle->AttachEffect(pkEffect);

```

After the scene graph is processed by the culler and the triangle is visible, the potentially visible set is passed to the renderer for drawing. Because `TextureEffect` is a single-pass effect and `pkTriangle` has only one effect attached to it with no dynamic lights, the drawing of the triangle requires only one pass.

To make it clear that *multitexture* and *multipass* are different concepts, drawing with multiple textures can be done in a single pass. The class `MultitextureEffect` is an example that allows multiple textures. This class is *incomplete* in that it cannot handle all possible combinations of textures and blending effects. This is one of those cases where there are a lot of possibilities that you would want to generate using shader stitching. The class was built to illustrate single-pass multitexturing; it handles three different combinations of two textures, each managing a 2D image.

The Cg vertex program that passes through two sets of 2D texture coordinates is `T0d2T1d2PassThroughVProgram.cg`. The letter T stands for “texture.” The letter d stands for “dimension.” Both d letters are followed by the number of dimensions for the texture image, in this case two for both textures.

The Cg pixel programs are `T0s1d0T1s1d1.cg`, `T0s1d0T1s2d0.cg`, and `T0s1d0T1s3d1.cg`. The naming conventions are the following. The letter T stands for “texture.” The letter s stands for “source-” blending function. The number immediately after the s is the integer assigned to the enumeration of `AlphaState::SrcBlendMode`. In the three sample programs, a value 1 means source-blending function `AlphaState::SBF_ONE`, a value 2 means source-blending function `AlphaState::SBF_DST_COLOR`, and a value 3 means source-blending function `AlphaState::SBF_ONE_MINUS_DST_COLOR`. The letter d stands for “destination-” blending function. The number immediately after the d is the integer assigned to the enumeration of `AlphaState::DstBlendMode`. In the three sample programs, a value 0 means destination-blending function `AlphaState::DBF_ZERO` and a value 1 means destination-blending function `AlphaState::DBF_ONE`. The prefix `T0s1d0` is always the same for the pixel programs. This is just to remind you that you may think of the first texture as being in “replace mode,” where the color buffer is overwritten with the first texture’s colors. The next block of T, s, d values specifies how the second texture is blended with the first. Keep in mind, though, that this is all happening in one pass; that is, the first and second textures are combined by the pixel program and then written to the color buffer. In summary, the blending is

```
T0s1d0T1s1d1: C0 + C1 // hard additive, clamped to [0,1]
T0s1d0T1s2d0: C0 * C1 // multiplicative
T0s1d0T1s3d1: (1-C0)*C1 + C0 // soft additive
```

where `C0` is a color from texture `T0` and `C1` is a color from texture `T1`.

The application code block that created a `TextureEffect` object can be modified for multitexturing:

```
// The root node of the scene.
m_spkScene = WM4_NEW Node;

// Create a single triangle whose vertices have positions and
// two sets of 2D texture coordinates.
Attributes kAttr;
```

```

kAttr.SetPChannels(3);
kAttr.SetTChannels(0,2); // unit 0
kAttr.SetTChannels(1,2); // unit 1
VertexBuffer* pkVBuffer = WM4_NEW VertexBuffer(kAttr,3);
pkVBuffer->Position3(0) = Vector3f(1.0f,0.0f,0.0f);
pkVBuffer->Position3(1) = Vector3f(0.0f,1.0f,0.0f);
pkVBuffer->Position3(2) = Vector3f(0.0f,0.0f,1.0f);
pkVBuffer->TCoord2(0,0) = Vector2f(0.0f,0.0f);
pkVBuffer->TCoord2(0,1) = Vector2f(1.0f,0.0f);
pkVBuffer->TCoord2(0,2) = Vector2f(0.0f,1.0f);
pkVBuffer->TCoord2(1,0) = Vector2f(0.5f,0.5f);
pkVBuffer->TCoord2(1,1) = Vector2f(1.0f,0.5f);
pkVBuffer->TCoord2(1,2) = Vector2f(0.5f,1.0f);
IndexBuffer* pkIBuffer = WM4_NEW IndexBuffer(3);
int* aiIndex = pkIBuffer->GetData();
aiIndex[0] = 0; aiIndex[1] = 1; aiIndex[2] = 2;
TriMesh* pkTriangle = WM4_NEW TriMesh(pkVBuffer,pkIBuffer);
m_spkScene->AttachChild(pkTriangle);

// Create a single-pass multitexture effect.
Effect* pkEffect = WM4_NEW MultitextureEffect(2);
pkEffect->SetImageName(0,"MyImage0");
pkEffect->SetImageName(1,"MyImage1");
pkTriangle->AttachEffect(pkEffect);

// Access the alpha blending state to select the shader
// for blending the two textures.
AlphaState* pkAState = pkEffect->GetBlending(1);

// For hard additive C0+C1:
pkAState->SrcBlend = AlphaState::SBF_ONE;
pkAState->DstBlend = AlphaState::DBF_ONE;
pkEffect->Configure();

// Or for multiplicative C0*C1:
pkAState->SrcBlend = AlphaState::SBF_DST_COLOR;
pkAState->DstBlend = AlphaState::DBF_ZERO;
pkEffect->Configure();

// Or for soft additive C0+C1:
pkAState->SrcBlend = AlphaState::SBF_ONE_MINUS_DST_COLOR;
pkAState->DstBlend = AlphaState::DBF_ONE;
pkEffect->Configure();

```

The `Configure` function call is what `MultitextureEffect` uses to determine which vertex and pixel shaders to use. The shader programs must already exist.

EXERCISE 4.2 Create more Cg programs to be used by the `MultitextureEffect` class for blending two textures. Create at least one program that uses three textures. Modify the `SampleGraphics/Multitextures` sample application to test your programs. ■

EXERCISE 4.3 Modify `MultitextureEffect::Configure` to stitch together separate shader programs. The alpha-state parsing code will not change, but instead of creating a file name, you should load the separate Cg programs, create a new Cg program combining them, save it to disk, shell out to the command line, and run NVIDIA's Cg compiler to create the assembly text programs. The name of the combined file can be structured similarly to the current system. Better yet, include the Cg Runtime environment in your application. After creating the combined Cg program, compile it using Cg Runtime (i.e., do not shell out to the command line) and insert the compiled result directly into the shader catalogs, thus bypassing the performance hit by shelling out to disk. ■

4.6.2 SINGLE-EFFECT, MULTIPASS DRAWING

The `ShaderEffect` class supports multipass drawing. The first pass draws the geometric primitive into the color buffer. Additional passes are blended into the color buffer according to the alpha blending state maintained by the effect object. This allows you to quickly obtain a lot of interesting effects. The drawback is that each additional pass requires the graphics system to re-rasterize the primitive, which can be quite expensive.

A multipass drawing operation can occur even if your geometric primitive has a single-pass effect attached to it. The dynamic lighting system automatically creates a `LightingEffect` object when lights affecting the geometric primitive are present in the scene. The `LightingEffect` class itself is designed to support a single-pass drawing with multiple lights as well as multipass drawing, but the current implementation allows only multipass. This class has a `Configure` function that parses the lights managed by the `LightingEffect` and determines which shader programs to use. This mechanism is similar to the one used in the class `MultitextureEffect`.

A sample code block for an application is shown next.

```
// The root node of the scene.
m_spkScene = WM4_NEW Node;

// Create a single triangle whose vertices have positions and
// normals (for dynamic lighting).
Attributes kAttr;
kAttr.SetPChannels(3);
```

```

kAttr.SetNChannels(3);
VertexBuffer* pkVBuffer = WM4_NEW VertexBuffer(kAttr,3);
pkVBuffer->Position3(0) = Vector3f(1.0f,0.0f,0.0f);
pkVBuffer->Position3(1) = Vector3f(0.0f,1.0f,0.0f);
pkVBuffer->Position3(2) = Vector3f(0.0f,0.0f,1.0f);
pkVBuffer->Normal3(0) = Vector3f(0.577f,0.577f,0.577f);
pkVBuffer->Normal3(1) = pkVBuffer->Normal3(0);
pkVBuffer->Normal3(2) = pkVBuffer->Normal3(0);
IndexBuffer* pkIBuffer = WM4_NEW IndexBuffer(3);
int* aiIndex = pkIBuffer->GetData();
aiIndex[0] = 0; aiIndex[1] = 1; aiIndex[2] = 2;
TriMesh* pkTriangle = WM4_NEW TriMesh(pkVBuffer,pkIBuffer);
m_spkScene->AttachChild(pkTriangle);

// Create an ambient light.
Light* pkALight = WM4_NEW Light(Light::LT_AMBIENT);
<set various light parameters>;

// Create a directional light.
Light* pkDLight = WM4_NEW Light(Light::LT_DIRECTIONAL);
<set various light parameters>;

pkTriangle->AttachLight(pkALight);
pkTriangle->AttachLight(pkDLight);

// This call will cause the Geometry portion of pkTriangle to
// create a LightingEffect object and attach the two lights
// to it.
pkTriangle->UpdateRS();

```

The UpdateRS call will cause the Geometry base class to create a LightingEffect object and attach the two lights to it. It also calls LightingEffect::Configure, which generates the shader file names from the light types. The ambient light causes the configuration to generate the name v_L1a.ext.wmsp, where ext is one of dx9, ogl, or sft. The directional light causes the configuration to generate the name v_L1d.ext.wmsp with the same choices for ext. The Renderer::Draw function, when applied to the geometric primitive, will initiate a multipass drawing operation. The ambient lighting is applied first, the directional lighting second.

The vertex shader programs for lighting are in the file Lighting.cg. This file contains helper functions for each of the light types (ambient, directional, point, spot). They were also constructed to allow you to easily build shaders that handle multiple lights. One such function is already in Lighting.cg and handles an ambient and a directional light.

EXERCISE 4.4 Create more Cg programs to be used by the `LightingEffect` class for single-pass lighting. Modify the `SampleGraphics/Lighting` sample application to test your programs. ■

EXERCISE 4.5 Modify `LightingEffect::Configure` to stitch together separate shader programs. The alpha-state parsing code will not change, but instead of creating a file name, you should load the separate Cg programs, create a new Cg program combining them, save it to disk, shell out to the command line, and run NVIDIA's Cg compiler to create the assembly text programs. The name of the combined file can be structured similarly to the current system. Better yet, include the Cg Runtime environment in your application. After creating the combined Cg program, compile it using Cg Runtime (i.e., do not shell out to the command line) and insert the compiled result directly into the shader catalogs, thus bypassing the performance hit by shelling out to disk. ■

4.6.3 MULTIPLE-EFFECT DRAWING

The Wild Magic scene graph system allows you to render as many effects as you like for a geometric primitive. You create a geometric primitive, create multiple effects, and attach them all to the primitive. For example, the single-pass multitexture effect can be performed as a multipass operation with two texture effects (not that you would want to do this due to the lower performance).

```
// The root node of the scene.
m_spkScene = WM4_NEW Node;

// Create a single triangle whose vertices have positions and
// 2D texture coordinates.
Attributes kAttr;
kAttr.SetPChannels(3);
kAttr.SetTChannels(0,2); // unit 0
VertexBuffer* pkVBuffer = WM4_NEW VertexBuffer(kAttr,3);
pkVBuffer->Position3(0) = Vector3f(1.0f,0.0f,0.0f);
pkVBuffer->Position3(1) = Vector3f(0.0f,1.0f,0.0f);
pkVBuffer->Position3(2) = Vector3f(0.0f,0.0f,1.0f);
pkVBuffer->TCoord2(0,0) = Vector2f(0.0f,0.0f);
pkVBuffer->TCoord2(0,1) = Vector2f(1.0f,0.0f);
pkVBuffer->TCoord2(0,2) = Vector2f(0.0f,1.0f);
IndexBuffer* pkIBuffer = WM4_NEW IndexBuffer(3);
int* aiIndex = pkIBuffer->GetData();
aiIndex[0] = 0; aiIndex[1] = 1; aiIndex[2] = 2;
TriMesh* pkTriangle = WM4_NEW TriMesh(pkVBuffer,pkIBuffer);
m_spkScene->AttachChild(pkTriangle);
```

```

Effect* pkEffect = WM4_NEW TextureEffect("MyImage0");
pkTriangle->AttachEffect(pkEffect);
pkEffect = WM4_NEW TextureEffect("MyImage1");
pkTriangle->AttachEffect(pkEffect);

// Access the alpha blending state to select the shader
// for blending the two textures.
AlphaState* pkAState = pkEffect->GetBlending(0);

// For hard additive C0+C1:
pkAState->SrcBlend = AlphaState::SBF_ONE;
pkAState->DstBlend = AlphaState::DBF_ONE;

// Or for multiplicative C0*C1:
pkAState->SrcBlend = AlphaState::SBF_DST_COLOR;
pkAState->DstBlend = AlphaState::DBF_ZERO;

// Or for soft additive C0*C1:
pkAState->SrcBlend = AlphaState::SBF_ONE_MINUS_DST_COLOR;
pkAState->DstBlend = AlphaState::DBF_ONE;

```

The texture effect using MyImage0 will be drawn first. The texture effect using MyImage1 is drawn second and blended with the first according to your choice of alpha blending state. The multipass rasterizes the triangle twice, so it is less efficient than the single-pass multitexture that rasterizes once.

4.7 SCENE GRAPH COMPILERS

As I mentioned in Section 4.1, a choice must be made whether to organize objects by spatial coherency or by render-state coherency. I presented the reasons why I believe the spatial organization is the main criterion and why render-state sorting is easily applied after a culling pass. Regardless, I have seen some heated debates on the game developer forums and in newsgroups where the premise is that organization by render state is better. Adding weight to this, I have also heard postmortems for a couple of commercial games. The use of scene graphs was criticized for performance reasons (among others). The term *scene graph* was used as if it had a single agreed-upon definition or mechanism. I will even venture to say that the term was used in a mystical sense, that somehow this single mechanism was supposed to be the end-all solution to graphics development in games, yet this mechanism did not live up to its expectations.

To paraphrase a statement made by one of the designers for Java3D regarding a scene graph: It's just a data structure. As with any data structure in computer science, it is important to know *how* to use that structure, but it is equally as important to know *what* the data structure was designed for and *when not to use it*, at least in the form that it was built.

When I worked at Numerical Design, Ltd. (now part of Emergent Game Technologies) developing NetImmerse (now Gamebryo) in the late 1990s, one of the first games to ship using that engine was *Prince of Persia 3D* from Red Orb Entertainment. Given the limited power of graphics hardware at that time (the game ran on 3dfx Voodoo cards), the Red Orb developers and artists did an excellent job getting this game to run with what they had. The process was nontrivial because they licensed the binaries for a 3D character training and animation tool. This tool insisted on managing all the transformations in the characters, but so did NetImmerse's scene graph management system. Red Orb did manage to get access to the animation tool source code and was able to remove the redundancy, and the game ran at real-time rates. What surprised many of us, though, was that the game was actually shipped using NetImmerse scene graphs, stored as files with the extension `.nif`.

I also recall responding to a post to the Usenet newsgroup, `comp.graphics.algorithms`, where the original poster wanted to know what the NIF file format was.³ This was for the popular game *Morrowind* (one of the Elder Scrolls games) from Bethesda Softworks, LLC. Apparently, NIF files were also shipped with this game. I find this surprising as well.

When designing the scene graph data structure in NetImmerse, the main goal was to mimic the organization provided by the data structures in the 3D Studio Max modeling package. Realizing that game developers would build their data sets and characters in such a package, we would have to export that data to a similar format to preserve things such as spatial locality. More important at the time, we had to preserve the animation information in articulated characters. The NIF files were not really a new “file format”; rather, they represented the current state of the scene graph data structures (nodes, geometric primitives, lights, cameras, etc.). We exported from 3D Studio Max to NIF files and then loaded the NIF files for development and testing. It was not our intent that games would actually ship with these. The data structures were intended for development, not for deployment.

A modeling package allows you to create and modify 3D objects any way an artist desires. During this process, you will find that an artist might place an object at one location in the modeling package's scene graph organization, but then later move it somewhere else in the scene. Adding and removing nodes in the scene is a relatively trivial operation for the artist. It is very easy to attach and detach parts of a scene,

3. I found the post by an advanced search on Google groups. The author name is “Dave Eberly”; the subject line is “NetImmerse File Format”; the original poster is “C. Smith”; and the date of the post is Monday, March 29, 2004, 4:20 p.m.

a very dynamic and flexible system for artists to use. Flexibility does not come for free. As with most things in computer science, you have trade-offs to consider. To remain flexible, the nodes and other objects in a scene are dynamically allocated and deallocated as needed. The final scene graph is invariably a collection of memory blocks scattered about the heap. When you process such a scene graph in the game application, you have memory fragmentation and you lose cache coherence because of the large jumps in memory needed to traverse from one node to another. The scene graph is convenient for development, but it is not necessarily ready for a graphics system (or a physics system or a 3D audio system).

The missing step here is a system that takes scene graphs created during development, either by exporting from a modeling package or created procedurally, and processes those scenes according to criteria for optimization. The output of such a system can be objects that have no aspect of scene graph management. More important, the output should be tailored for the target platform, whether that be a PC, a game console, or even an embedded device such as a mobile phone with video capabilities. The list of optimizations can be quite enormous and specific to your own application's needs. I like to think of the tools in the system in terms of *compilers*, each compiler having a scene graph input and producing an optimized output.

4.7.1 A SCENE GRAPH AS AN EXPRESSION

The developmental view of a scene graph is that it is a dynamic data structure. In compiler terms, you may also think of it abstractly as an *expression*. Sometimes the scene graph is *static*, but in many cases it is *dynamic*. Some examples are presented here to illustrate how to think of the scene graph as an expression and how to write a compiler to convert it to something optimized.

EXAMPLE 4.1

Consider a complicated level of a game that has lots of static geometric objects. During development, you allow the artists the flexibility to create, modify, and move these objects around the environment. All the objects are stored in one or more (dynamic) scene graphs. When it comes time to load these scene graphs and display them, the graphics engine wants to know what is potentially visible and what is not. The hierarchical culling of a scene graph allows you to construct the potentially visible set for each frame of drawing. Suppose that your level is indoors and you know the game character is in a certain room of the level. During development you can precompute the visibility, determining the largest set of potentially visible objects that the game character can see from that room. The static scene graphs for the level may be preprocessed to produce a collection of geometric primitives that are loaded from disk, and then the graphics engine just draws them without any culling tests. In this example, the scene graphs representing the static geometry are expressions that are run through a compiler that produces a collection of geometric primitives. ■



Figure 4.9 A rendering of the skinned biped object from the skinned biped sample application.

EXAMPLE
4.2

Another common example of a scene graph that is convenient for modeling and development, but should probably be optimized, is an articulated biped character. The sample application `SampleGraphics/SkinnedBiped` loads the components of the scene graph and assembles them into a single scene graph. The rendered object is shown in Figure 4.9.

The scene graph structure is summarized next. Each line of text corresponds to a node in the scene graph. The indentation denotes parent-child relationships.

```
Node<"Biped">
  Node<"Pelvis">
    Node<"Spine">
      Node<"Spine1">
        Node<"Spine2">
          Node<"Spine3">
            Node<"Neck">
              Node<"Head">
                Node<"L Clavicle">
                  Node<"L UpperArm">
                    Node<"L Forearm">
                      Node<"L Hand">
                        TriMesh<"L Arm">
                        TriMesh<"Hair">
```

```

        Node<"R Clavicle">
            Node<"R UpperArm">
                Node<"R Forearm">
                    Node<"R Hand">
                        TriMesh<"R Arm">
                            TriMesh<"Face">
Node<"L Thigh">
    Node<"L Calf">
        Node<"L Foot">
            Node<"L Toe">
                TriMesh<"L Shoe">
                    TriMesh<"L Leg">
                        TriMesh<"L Ankle">
Node<"R Thigh">
    Node<"R Calf">
        Node<"R Foot">
            Node<"R Toe">
                TriMesh<"R Shoe">
                    TriMesh<"R Leg">
                        TriMesh<"R Ankle">
TriMesh<"Shirt">
TriMesh<"Pants">

```

Most of the Node objects have keyframe controllers attached. The TriMesh objects have skin controllers attached. They also have materials attached to produce the colors you see in Figure 4.9. The TriMesh objects also have normal vectors for dynamic lighting. The scene is rendered using a directional light.

The biped has a lot of nodes in the scene, all occurring because this is the way the modeling package represented the character during its construction. The nodes are all dynamically allocated when the Wild Magic scene graph file is loaded from disk. If the topology of this scene will not change during the application run time, then it is possible to store the scene in a single block of contiguous memory, hopefully avoiding cache misses due to jumping around in memory when the scene graph is used. By storing the biped in a contiguous block of memory, the deletion of the biped will lead to at most one hole in the heap, thereby reducing the potential memory fragmentation that the scene graph will cause when it is deleted.

For a single-processor machine running a single thread, the simplest compaction scheme will store the relevant node information in an array of NodeInformation objects. The order of the elements can be the one implied by a depth-first traversal of the scene. For the most part, the important node information consists of the local and world transformation storage and the keyframe controller data. The triangle mesh vertex and index buffers can also be stored in an array. Since we do not even want

to dynamically allocate the `MaterialState` objects used for coloring the meshes, the material information would be stored with the meshes.

Also notice that if you have no plans ever to modify the vertex or index buffers for the meshes, you could provide two storage locations per mesh for pointers, one pointer to a vertex buffer and one pointer to an index buffer. At file load time, you dynamically allocate the vertex and index buffers in system memory, request that these resources be loaded to VRAM, and then deallocate the vertex and index buffers that are in system memory. The handles to the buffers that the graphics system gives you can be stored in those storage locations that used to contain the system-memory buffer pointers. These handles are used later to let the graphics system know which buffers to use when drawing the biped.

If your application is running in a multithreaded environment or on a machine with multiple processors, you could organize subtrees of the scene graph into a few blocks for the purpose of transformation (during an `UpdateGS` call). For example, you can store transformations in three separate blocks, one for the “Spine” subtree, one for the “L Thigh” subtree, and one for the “R Thigh” subtree. The transformations in the three blocks may be updated by three separate threads dedicated to computing transformations, thus allowing you to do the geometric updates in parallel.

From the perspective of the library classes, you will implement a new class called `SkinnedBiped` to store and represent the contiguous block of memory. A compiler is written that will take as input a skinned biped scene graph of the form shown and produce a `SkinnedBiped` object as output. ■

EXAMPLE 4.3

Game development and programming is typically done on one platform, such as a PC, but the testing is done on each target platform, such as a PC, Macintosh, or one or more game consoles. What runs well on a PC might not necessarily run well on another platform. Moreover, the data storage requirements might vary. For example, the Intel-based machines use little endian byte order, but PowerPC-based machines use big endian byte order. If your scene graph management system stores everything in little endian order, and if you ship files containing this data for a real application, a big endian system must swap the byte order when loading the data. It is better to ship data sets that are optimized for the target platform. You may very well build a scene graph compiler to assist in saving files to disk. Its job would be to swap byte order when storing the data to disk to be used by a big endian machine but to do nothing when storing the data to disk to be used by a little endian machine.

The storage of image data has similar issues, even on a single platform. If the graphics system expects image data in a specific format and your run-time system loads data in a different format and has to convert it to what the graphics system wants, you probably want to reconsider factoring the conversion code out of your run-time system. Any time any data has to be repackaged in the actual application, there is a good chance you can improve performance by doing the repackaging during development (or even during installation, if need be, when you know the system the

application will run on). Ideally, the run-time environment for the application should contain as little “development support code” as possible. ■

EXAMPLE
4.4

The culling of objects in a scene graph may itself be considered a scene graph compiler. The input to `Culler::ComputeVisibleSet` is a scene graph. The output is a potentially visible set. For a game environment where the camera remains fixed for multiple frames, it is beneficial to cull once for those multiple frames rather than culling per frame. For cell-based visibility, it is even possible that `ComputeVisibleSet` is only ever used during development to create the potentially visible sets per cell. The actual application would then load per cell the list of objects that should be sent to the renderer without the application ever culling once. ■

4.7.2 SEMANTICS OF COMPILETIME

In the examples considered so far, you have metaknowledge of the scene graphs that need to be optimized. Unless you have added explicit information into the scene graph classes, a compiler has no knowledge that a particular scene graph represents static geometry. The compiler to convert a skinned biped scene graph to a `SkinnedBiped` object could attempt to validate its input by checking the topology of the scene, but if your tool chain is set up so that you only ever feed skinned biped objects to this compiler, there is no need to validate.

More complicated examples, though, require you to assist the compiler to achieve the *semantics* you want the scene graph to have. The analogy of *syntax* is knowing how to traverse a scene and manipulate its parts. Semantics tell you how to interpret what the scene means. The simplest way to provide semantics is to use *node tags*. Each node in the scene graph is tagged with information designed to be read by the compiler; the tag tells the compiler how to handle that node.

Tagging can be quite simple. For example, some modeling packages export a viewable skeleton for a biped in addition to the meshes that form the biped’s skin. This skeleton is for the artist’s convenience and is not relevant to how you intend on using the biped in an application. The final exported data set should not have that skeleton, but a generic exporter will notice that the skeleton is just as valid an object to export as any other. The way around this is to have the artist insert textual information into an object’s name string (a feature provided by nearly all modeling packages) indicating that the object should be skipped when exporting. The exporter (the compiler as it were) parses the scene graph, reading name strings and responding accordingly.

By the same token, tagging systems can be much more complicated.

EXAMPLE
4.5

Visibility determination for an indoor level may be automated by using rooms and portals between rooms. The level itself may be represented by a scene graph (or a subtree of a scene graph). Each level consists of a collection of rooms. Each opening between adjacent rooms is a portal through which you can see the adjacent room

from the one you are currently in. The drawing pass will not proceed according to a depth-first traversal of the scene graph representing the level. Instead, the traversal path depends on the room-portal graph and where the camera is currently located.

Given a level, it is a very difficult problem to automatically compute the room-portal graph. It is better to have the artists structure the level within the modeling package to conform to certain requirements. For example, if the root of the scene graph represents the level, you might require that all the rooms be children of the root. Each room has geometric components that define the floor, ceiling, and walls. Each room also has contents (chairs, tables, light fixtures, and so on) that are not part of the geometry that defines the room boundaries. Doorways and windows between adjacent rooms are what define the room-portal graph.

If the level is correctly structured, the artist can add node tags with information that is used by a semiautomatic portalizing tool. The root node is tagged as the level itself. Each child is tagged as a room. The floor, ceiling, and wall geometry are tagged with information indicating that the portalizing tool should use these to define the region of space containing a room. The bounding region information is used to determine which room currently contains the camera so that a drawing pass may be properly initiated. The doorway and window geometry must also be tagged as portals, including information about the two rooms sharing that portal. An exporter can then convert the modeling package's scene graph to your engine's scene graph representation. The portalizing tool processes this scene and replaces the nodes with the appropriate objects in your class hierarchy.

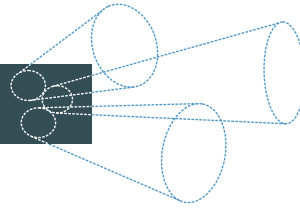
For example, the top-level node for a room-portal scene is of type `ConvexRegionManager`. If the exporter converted the modeling package scene to something that has a `Node` for the root and is tagged as "Dungeon Level" (for example), the portalizing tool will replace that `Node` by a `ConvexRegionManager` object. The `Node` objects tagged as rooms are replaced by `ConvexRegion` objects. The tagged room geometry (walls, floors, ceilings) in the scene may be collected into a single set of points, and a convex hull is computed to be used as the bounding region for the room. Finally, each doorway or window tagged as a portal has information about the two rooms sharing them. This is used to create `Portal` objects in the scene. Thinking of the portalizer as a scene graph compiler, the input is a scene graph without portal-system objects, but with various tagged objects. The output is a scene graph with portal-system objects and with the tagged objects removed (including their tags). ■

EXAMPLE 4.6

An example similar to the portalizer tools is a compiler that also processes the scene graph output by an exporter. Support for automatic pathfinding can be added to a scene. Just as in a portal system, the environment consists of a collection of rooms interconnected by doorways. Each room is bounded by walls, a floor, and a ceiling. The rooms also contain obstacles around which a game character must navigate. Rather than rely on a generic collision detection system to assist in navigation, you can create *blueprints* of the rooms, which may then be used by an automatic pathfinding system. The level, rooms, and obstacles need tagging. The tagged (vertical) walls are projected

onto ground level to form a bounding polygon for the room. The tagged obstacles are also projected to form bounding polygons for the obstacles. The blueprint consists of polygons for rooms and obstacles, with gaps in the room polygons to indicate doorways between rooms. An abstract graph may be built from the blueprints and used for visibility purposes. One point in the blueprint is reachable from another point if there is a path connecting the two that does not pass through any obstacle or room wall. In a polygonal environment, these paths are polylines whose vertices are polygon vertices. I cover this system in detail in Section 8.5. However, it is up to you to write the compiler to postprocess an exported scene. ■

These examples should make it clear that it is not feasible to think of a scene graph compiler as a black box that can process any scene you throw at it. Your tool chest will have quite a collection of compilers, all designed to meet your specific needs.



CONTROLLER-BASED ANIMATION

I use a very general definition for the word *animation*, using it to describe the process of controlling any time-varying quantity in a scene graph. The classic setting is character animation, where an articulated object has joints that change position and orientation over time. The quantities to be controlled are the local transformations at the joints. Two standard approaches to animating a character are discussed next.

This chapter covers some basic animation controllers. Section 5.1 is about *keyframe animation*. When implemented for 3D characters, an artist is required to build a character in various poses; each pose is called a *keyframe*. Each keyframe changes the local positions and local orientations of the nodes in the hierarchy. When it comes time to animate the character, the poses at the times between keyframes are computed using interpolation. A flexible method for interpolating the local translations uses Kochanek-Bartels splines, discussed in detail in Section 11.7. The interpolation of local rotations is more complex, but the concept of splines still applies. The idea is to represent the rotations as quaternions and then interpolate the quaternions. The smooth interpolation of quaternions is a somewhat technical concept. Details may be found in Section 17.2.

One potential problem with keyframe animation is that the local transformations at the nodes are interpolated in a relatively independent way. Interpolation at one node is performed independently from interpolation at another node, which can lead to artifacts, such as the stretching of character components that normally are considered to be rigid. For example, the local translations of a shoulder node and elbow node are interpolated independently, but the length of the arm from shoulder

to elbow should be constant. The interpolations do not guarantee that this constraint will be satisfied.

The game content can have a large quantity of keyframes, which requires a lot of memory to store. Reductions in memory usage are possible via *keyframe compression*. This is the topic of Section 5.2.

Keyframe animation uses data that is precomputed by an artist. If the animation must occur dynamically in unexpected ways, an alternative method is to use *inverse kinematics*. Constraints are placed at the various nodes—constraints such as fixed lengths between nodes or rotations restricted to planes and/or with restricted ranges of angles. The only interpolation that needs to occur is at those nodes with any degree of freedom. For example, an elbow node and wrist node have a fixed length between them, and both nodes have rotations restricted to planes with a fixed range of angles. A hand node is attached to the wrist and has three degrees of freedom (the components of local translation). The hand can be moved to some location in space; the wrist and elbow must follow accordingly, but with the mentioned constraints. Section 5.3 is about inverse kinematics.

Animating the nodes corresponding to the joints of a character is one thing, getting the surface of the character to move properly with the joints is another. Section 5.4 talks about *skinning*—establishing a set of bones connecting the joints and assigning the vertices of the mesh representing skin, clothing, and other quantities to various bones for weighting purposes. As the bones move, the vertices will change according to their weights.

Section 5.5 is about time-varying vertices of a mesh, or *vertex morphing*. As the vertices move about, the mesh morphs into various forms. There are a few ways to control the vertices. In Wild Magic, I have a class called `MorphController` that manages a collection of poses and morphs from one pose to the next. It is also possible to have a morphing controller for which each vertex has its own morphing function.

The final topic is Section 5.6 on *point systems* and *particle systems*. Points are displayed as single pixels on the screen, but particles have size and are typically represented as billboarded screen polygons. Both systems have a physics aspect to them—controlling the position (points and particles) and the orientation (particles) over time due to forces and torques applied to them.

Now a brief mention of what this chapter is *not* about. Nothing is discussed in this chapter about how to actually build physically realistic animations. That topic is quite complex and could fill a large book by itself. Regarding particle physics, a few companies have ventured into producing *physics engines* to provide for realistic motions and realistic interactions between objects in the world. Although correct physics is a very important topic, this chapter describes only how to process the animation data that was already constructed by an artist through a modeling package, by motion capture, or by other procedural means.

Support for controllers in Wild Magic are provided by the base class `Controller`, which has data members for specifying the time range over which the animation occurs. It also has members for specifying how the times are to be interpreted; the units of time may vary based on the modeling package used to generate the controlled

data. The animation time may be clamped, repeated, or cycled (mirror-repeated), modes that are analogous to those found in texture coordinate modes. The `Object` class provides the ability to attach controllers to objects and detach controllers from objects. The `Controller` class has a virtual function that supports dynamic updates during the program execution. The updating essentially uses the input time and allows the controller to interpolate data or perform whatever action is necessary to modify its state so that it represents what is expected at the input time. Each `Controller`-derived class overrides the virtual update function as needed.

5.1 KEYFRAME ANIMATION

The standard implementation of keyframe animation involves interpolating positions, orientations, and scales as separate *channels*. This requires the nodes in a hierarchy to store the channels separately. If you were to design a transformation hierarchy that stores general matrices at the nodes, and if you wanted to interpolate a pair of matrices to produce visually intuitive results, you would have to factor the matrices into positions, orientations, and scales. This problem is ill-posed; see Section 17.5.

5.1.1 INTERPOLATION OF POSITION

In Wild Magic, the `KeyframeController` class stores the positions, orientations, and scales separately. A sequence of keyframe positions is $(t_i, \mathbf{P}_i)$ for $0 \leq i < n$. The times are assumed to be ordered, $t_0 < t_1 < \dots < t_{n-1}$, but not necessarily uniformly spaced. If the query time is $t \in [t_0, t_{n-1}]$, you must first locate the pair of keys whose times bound the query time; that is, search for the index i for which $t_i \leq t < t_{i+1}$. A *normalized time* is computed,

$$u = \frac{t - t_i}{t_{i+1} - t_i} \quad (5.1)$$

and the position keys are interpolated by

$$\mathbf{P} = (1 - u)\mathbf{P}_i + u\mathbf{P}_{i+1} \quad (5.2)$$

Knowing that the interpolation is going to happen at real-time rates, a linear search for the index i can require enough time that it is noticeable when you profile. You may take advantage of time coherency. If i_{last} is the index computed on the last query for interpolation, rather than starting the linear search at index 0, start it at i_{last} . A general linear search over n items is of order $O(n)$. If you start the search at the last index, you expect an $O(1)$ search. In most cases, i_{last} will be the index used for the current search, but sometimes will be $i_{\text{last}} + 1$. If your keyframe animations will include the cycle mode (mirrored-repeat mode), then you would search indices

smaller than i_{last} when the cycle mode is in the reversed direction of time. Still the expected search is order $O(1)$.

5.1.2 INTERPOLATION OF ORIENTATION

A sequence of keyframe orientations is (t_i, q_i) for $0 \leq i < n$, where q_i is a unit quaternion that represents a rotation. The times are assumed to be ordered, $t_0 < t_1 < \dots < t_{n-1}$, but not necessarily uniformly spaced. Although I have used the same n for the number of keys, a keyframe controller may have different numbers of keyframes for positions, orientations, and scales. The `KeyframeController` class allows this, and it allows you to specify a common set of times if that is what your animation uses.

The normalized time is computed using Equation (5.1). The orientation keys are interpolated using the *slerp* function for quaternions,

$$q = \text{slerp}(t; q_i, q_{i+1}) = \frac{q_i \sin((1-t)\theta_i) + q_{i+1} \sin(t\theta_i)}{\sin \theta_i} \quad (5.3)$$

where θ_i is the angle between q_i and q_{i+1} . Treating the quaternions as 4-tuples, the dot product is $q_i \cdot q_{i+1} = \cos \theta_i$. See Section 17.2 for details on quaternions. This section also discusses the relationship between quaternions and rotations. It is possible to do the keyframe interpolation using rotation matrices, but at a cost of many more CPU cycles.

One problem with quaternions is that a single rotation/orientation is represented by two quaternions, q and $-q$. This can cause visual anomalies to occur when interpolating during keyframe animations. In fact, it has happened for me when exporting quaternions from 3dsmax. The problem is that you want to interpolate two consecutive quaternions as long as the angle between them is acute. If you have a sequence q_i for $0 \leq i < n$, you should preprocess them to guarantee the acute-angle condition. That is,

```
for (i = 1; i < n; i++)
{
    float dot = Dot(q[i-1], q[i]);
    if (dot < 0) // The angle is obtuse.
    {
        q[i] = -q[i];
    }
}
```

5.1.3 INTERPOLATION OF SCALE

A sequence of scales is (t_i, σ_i) for $0 \leq i < n$, where I assume $\sigma_i > 0$. As before, the number of scales is not required to be the number of positions or the number of

orientations. Moreover, it is not necessary to have all three types of keyframes. You can mix and match as needed.

The interpolations for positions and orientations were chosen in a “natural” manner. The natural manner for scales is geometric rather than algebraic:

$$\sigma = \sigma_i^{1-u} \sigma_{i+1}^u \quad (5.4)$$

for normalized time $u \in [0, 1]$. The interpolation is actually linear in the logarithm of scale:

$$\log(\sigma) = (1 - u) \log(\sigma_i) + u \log(\sigma_{i+1})$$

However, artists tend not to choose large scales, instead using them to tweak the way models appear. In this case, a linear interpolation of scale works fine and is what I implemented in Wild Magic:

$$\sigma = (1 - u)\sigma_i + u\sigma_{i+1} \quad (5.5)$$

Nonuniform scaling has been the curse of computer graphics and 3D models. I made the assumption that the scales are positive, but artists do tend to use negative scales to obtain reflections of objects. This is a curse, because if you have a triangle mesh with counterclockwise-ordered triangles when viewed from outside the mesh, and if you apply a transformation with a negative scaling factor, the reflected mesh has its triangle ordering reversed. You could detect this and attach a culling-state object to the mesh, asking for front-facing triangles to be culled, but I personally find it to be a pain having to deal with the reflections and negative scales.

Another problem with nonuniform scaling has to do with wanting to “bake” a parent’s world transformation into a child’s keyframe data. For example, suppose a child has a pair of keyframe positions $\mathbf{P}_0$ and $\mathbf{P}_1$. The interpolation produces the time-varying translation

$$\mathbf{T}_0(t) = (1 - t)\mathbf{P}_0 + t\mathbf{P}_1$$

for $t \in [0, 1]$. Suppose $\mathbf{X}$ is a point in the child’s coordinate system, and suppose $\mathbf{Y} = R_1 S_1 \mathbf{X} + \mathbf{T}_1$ is the transformation to world space, where R_1 is the parent’s world rotation, S_1 is the parent’s world (nonuniform) scales, and $\mathbf{T}_1$ is the parent’s world translation. Applying this during keyframe animation, we have

$$\begin{aligned} \mathbf{Y}(t) &= R_1 S_1 \mathbf{T}_0(t) + \mathbf{T}_1 = R_1 S_1 ((1 - t)\mathbf{P}_0 + t\mathbf{P}_1) + \mathbf{T}_1 \\ &= (1 - t)(R_1 S_1 \mathbf{P}_0 + \mathbf{T}_1) + t(R_1 S_1 \mathbf{P}_1 + \mathbf{T}_1) \end{aligned}$$

The parent’s world transformation is baked into the child’s position keyframes to obtain $\mathbf{P}'_i = R_1 S_1 \mathbf{P}_i + \mathbf{T}_1$. The baking might allow you to remove the parent node from the hierarchy if it has no other responsibilities than providing a world transformation for its children.

A slight extension of this construction shows that rotations and translations can be baked into the child's keyframes. The translational component is the same as was shown previously. If the orientation keyframes are the quaternions q_0 and q_1 , then $q(t)$ is defined by Equation (5.3). If α_1 is the quaternion corresponding to the parent's world transformation, then the matrix component after application of the world transformation is

$$\begin{aligned}\alpha_1 q(t) &= \alpha_1 \frac{q_i \sin((1-t)\theta_i) + q_{i+1} \sin(t\theta_i)}{\sin \theta_i} \\ &= \frac{(\alpha q_i) \sin((1-t)\theta_i) + (\alpha q_{i+1}) \sin(t\theta_i)}{\sin \theta_i}\end{aligned}$$

Using the algebraic properties of quaternions, the angle between αq_i and αq_{i+1} is θ . Think of α as rotating q_i and q_{i+1} on the unit 4D hypersphere—the rotation preserves angles. Thus, the baked keyframes are $q'_i = \alpha q_i$.

When you throw in nonuniform scaling, the baking does not work. It all comes down to the problem that you cannot generally take a product $S_0 R_0$ of a nonuniform scale matrix S_0 and a rotation matrix R_0 and refactor it as $S_0 R_0 = R_1 S_1$, where R_1 is a rotation and S_1 is a nonuniform (diagonal) scale matrix. The best you can do is obtain a symmetric matrix S_1 ; see Section 17.5 for details.

5.2 KEYFRAME COMPRESSION

Modeling packages that support animation make it difficult to extract the internal continuous representation of keyframe interpolation, but they make it relatively easy to sample the continuous representation to produce the keyframe data. What many developers have found is that avoiding the quirks of the modeling package export SDKs is preferred. Instead they choose to sample the animations at a constant frame rate (30 or 60 samples per second) and use the data as is, or when memory is scarce (such as on game consoles), reduce the number of keyframes through algorithmic means.

A simple method to reduce the memory usage is to store the samples using a 16-bit representation of the 32-bit floating-point values, which is a form of lossy data compression. For positional and scale data, the floating-point channels are not restricted to a particular interval. The first step is to compute the extreme values for a channel and then map the 32-bit floating-point values to 16-bit unsigned integers. Specifically, let the minimum of the channel be m_0 and let the maximum of the channel be m_1 . The transformation of the floating-point value $f \in [m_0, m_1]$ to a 16-bit unsigned integer i is

$$i = \left\lfloor 65,535 \left(\frac{f - m_0}{m_1 - m_0} \right) \right\rfloor$$

where $\lfloor x \rfloor$ is the floor function, which computes the largest integer smaller or equal to x . The transformed values are stored in memory as the keyframes. During the program execution, the 16-bit unsigned integers are transformed to 32-bit floating-point numbers and then interpolated as discussed in Section 5.1. The reverse transformation is

$$f = m_0 + \frac{(m_1 - m_0)i}{65,535}$$

Naturally, the divisions should be avoided by precomputing some values. For example, you would precompute $c = (m_1 - m_0)/65,535$ as a floating-point number so that the interpolator may compute $f = m_0 + ci$. The typical space-time trade-off is made here. You reduce your memory usage but pay the price by using extra time (uncompressing) to do the interpolation. For game consoles, memory is small and computing power is large, so the trade-off is a good one.

A more sophisticated method for reducing memory usage is to fit the samples with a B-spline curve. The goal is to construct a curve that is a good fit to the (unknown) internal representation that the modeling package uses, but the curve should have many fewer control points than the number of samples to which the curve is fit. The animation occurs by evaluating the B-spline curve at selected times. Regarding a good fit to an unknown internal representation, the artists have the final say. They should look at the animation based on the B-spline evaluation, visually compare it to what they see in the modeling package, and give a thumbs-up (or thumbs-down as the case may be). An algorithm for fitting samples with a B-spline curve is discussed here. This approach has been known to the CAD community for quite some time.

5.2.1 FITTING POINTS WITH A B-SPLINE CURVE

A set of keyframe samples is $\{(t_k, \mathbf{P}_k)\}_{k=0}^m$, where the t_k are the sample times and $\mathbf{P}_k$ are the sample data. The sample times must be increasing: $t_0 < t_1 < \dots < t_m$. The points can be in any dimension. For our needs, they are positions (3D) or unit-length quaternions (4D). In the latter case, the fitted curve is not guaranteed to be on the unit hypersphere in 4D, but each curve evaluation is normalized to produce a unit-length result. The normalization may be implemented using a fast inverse square root algorithm.

A fitted B-spline curve uses the normalized time $u \in [0, 1]$ of Equation (5.1). The mapping from t_k to u_k is an off-line process and the mapping from u_k to t_k during run time is of negligible cost. A *B-spline curve* is defined for a collection of $n + 1$ control points $\{\mathbf{Q}_i\}_{i=0}^n$ by

$$\mathbf{X}(u) = \sum_{i=0}^n N_{i,d}(u) \mathbf{Q}_i \quad (5.6)$$

The *degree* of the curve is d and must satisfy $1 \leq d \leq n$. The functions $N_{i,d}(u)$ are the *B-spline basis functions*, which are defined recursively and require selection of a sequence of scalars u_i for $0 \leq i \leq n + d + 1$. The sequence is nondecreasing; that is, $u_i \leq u_{i+1}$. Each u_i is referred to as a *knot* and the total sequence as a *knot vector*. The basis function that starts the recursive definition is

$$N_{i,0}(u) = \begin{cases} 1, & u_i \leq u < u_{i+1} \\ 0, & \text{otherwise} \end{cases} \quad (5.7)$$

for $0 \leq i \leq n + d$. The recursion itself is

$$N_{i,j}(u) = \frac{u - u_i}{u_{i+j} - u_i} N_{i,j-1}(u) + \frac{u_{i+j+1} - u}{u_{i+j+1} - u_{i+1}} N_{i+1,j-1}(u) \quad (5.8)$$

for $1 \leq j \leq d$ and $0 \leq i \leq n + d - j$. The *support* of a function is the smallest closed interval on which the function has at least one nonzero value. The support of $N_{i,0}(u)$ is clearly $[u_i, u_{i+1}]$. In general, the support of $N_{i,j}(u)$ is $[u_i, u_{i+j+1}]$. This fact means that locally the curve is influenced by only a small number of control points, a property called *local control*.

The main classification of the knot vector is that it is either *open* or *periodic*. If open, the knots are either *uniform* or *nonuniform*. Periodic knot vectors have uniformly spaced knots. The use of the term *open* is perhaps a misnomer since you can construct a closed B-spline curve from an open knot vector. The standard way to construct a closed curve uses periodic knot vectors. For the fitting of keyframe samples, we will restrict our attention to open and uniform knot vectors:

$$u_i = \begin{cases} 0, & 0 \leq i \leq d \\ \frac{i-d}{n+1-d}, & d+1 \leq i \leq n \\ 1, & n+1 \leq i \leq n+d+1 \end{cases} \quad (5.9)$$

The uniformity is important in guaranteeing a robust curve fit.

We consider the control points $\mathbf{Q}_i$ unknown quantities to be determined later. The control points are considered to be column vectors, and the collection of control points may be arranged into a single column vector

$$\hat{\mathbf{Q}} = \begin{bmatrix} \mathbf{Q}_0 \\ \mathbf{Q}_1 \\ \vdots \\ \mathbf{Q}_n \end{bmatrix} \quad (5.10)$$

Similarly, the samples $\mathbf{P}_k$ are considered to be column vectors, and the collection is written as a single column vector

$$\hat{\mathbf{P}} = \begin{bmatrix} \mathbf{P}_0 \\ \mathbf{P}_1 \\ \vdots \\ \mathbf{P}_m \end{bmatrix} \quad (5.11)$$

For a specified set of control points, the *least-squares error function* between the B-spline curve and sample points is the scalar-valued function

$$E(\hat{\mathbf{Q}}) = \frac{1}{2} \sum_{k=0}^m \left| \sum_{j=0}^n N_{j,d}(u_k) \mathbf{Q}_j - \mathbf{P}_k \right|^2 \quad (5.12)$$

The quantity $\sum_{j=0}^n N_{j,d}(u_k) \mathbf{Q}_j$ is the point on the B-spline curve at the scaled sample time u_k . The term within the summation on the right-hand side of Equation (5.12) measures the squared distance between the sample point and its corresponding curve point. The error function measures the total accumulation of squared distances. The hope is that we may choose the control points to make this error as small as possible. Although zero error would be great to achieve, this can happen only if the samples were chosen from a B-spline curve itself, which is not likely, but nevertheless we want to minimize the error.

The minimization is a calculus problem. The function E is quadratic in the components of $\hat{\mathbf{Q}}$, so it must have a global minimum that occurs when its gradient vector (the vector of first-order partial derivatives) is zero. The analogy you are most likely familiar with is a parabola that opens upward. The vertex of the parabola occurs where the first derivative is zero. The first-order partial derivatives are written in terms of the control points $\mathbf{Q}_i$ rather than in terms of the components of the control points. This allows us to manipulate vector-valued equations in a manner that is more conducive to solving the problem symbolically. The derivatives are

$$\begin{aligned} \frac{\partial E}{\partial \mathbf{Q}_i} &= \sum_{k=0}^m \left(\sum_{j=0}^n N_{j,d}(u_k) \mathbf{Q}_j - \mathbf{P}_k \right) N_{i,d}(u_k) \\ &= \sum_{k=0}^m \sum_{j=0}^n N_{i,d}(u_k) N_{j,d}(u_k) \mathbf{Q}_j - \sum_{k=0}^m N_{i,d}(u_k) \mathbf{P}_k \\ &= \sum_{k=0}^m \sum_{j=0}^n a_{ki} a_{kj} \mathbf{Q}_j - \sum_{k=0}^m a_{ki} \mathbf{P}_k \end{aligned} \quad (5.13)$$

where $a_{rc} = N_{c,d}(u_r)$, and for $0 \leq i \leq n$. Setting the partial derivatives equal to the zero vector leads to the system of equations

$$\mathbf{0} = \sum_{k=0}^m \sum_{j=0}^n a_{ik} a_{jk} \mathbf{Q}_j - \sum_{k=0}^m a_{ik} \mathbf{P}_k = \mathbf{A}^T \mathbf{A} \hat{\mathbf{Q}} - \mathbf{A}^T \hat{\mathbf{P}} \quad (5.14)$$

where $A = [a_{rc}]$ is a matrix with $n + 1$ rows and $m + 1$ columns. The matrix A^T is the transpose of A . This system of equations is in a familiar form of a least-squares problem. Recall that such problems arise when wanting to solve $A\mathbf{x} = \mathbf{b}$. If the system does not have a solution, the next best thing is to construct $\mathbf{x}$ so that $\|A\mathbf{x} - \mathbf{b}\|$ is as small as possible. The minimization leads to the *normal system* $A^T A \mathbf{x} = A^T \mathbf{b}$.

The matrix $A^T A$ is symmetric, a property that is desirable in the numerical solution of systems. Moreover, the matrix A is *banded*, which is a generalization of *tridiagonal*. A banded matrix has a diagonal with (potentially) nonzero entries. It has a contiguous set of *upper bands* and a contiguous set of *lower bands*, each band with (potentially) nonzero entries. All other entries in the matrix are zero. In our case the number of upper bands and the number of lower bands are the same, namely, $d + 1$. The bandedness is a consequence of the local control for B-spline curves (the supports of the B-spline basis functions are bounded intervals).

You might try inverting the coefficient matrix directly to solve Equation (5.14). That is, the equation $A^T A \hat{Q} = A^T \hat{P}$ implies

$$\hat{Q} = (A^T A)^{-1} A^T \hat{P}$$

The problem, though, is that the matrix inversion is ill-conditioned when the determinant of $A^T A$ is nearly zero. The ill-conditioning causes Gaussian elimination to have problems, even with full pivoting. As it turns out, the ill-conditioning is not typically an issue as long as you choose a B-spline curve with uniform knots. Indeed, this is the reason I mentioned earlier that we will constrain ourselves to uniform knots. Regardless, a different approach to solving the linear system is called for, both to minimize the effects of ill-conditioning and to take advantage of the bandedness of the matrix. Recall that Gaussian elimination to solve a linear system with an $n \times n$ matrix is an $O(n^3)$ algorithm. The solution to a linear system with a tridiagonal matrix is $O(n)$; the same is true for a banded matrix with a small number of bands relative to the size of the matrix.

The numerical method of choice for symmetric, banded matrix systems is the *Cholesky decomposition*. The book [GL93] has an excellent discussion of the topic. The algorithm starts with a symmetric matrix and factors it into a lower-triangular matrix times the transpose of that lower-triangular matrix. In our case, the Cholesky decomposition is

$$A^T A = G G^T$$

where G is lower triangular and G^T is upper triangular. The linear system is then $G G^T \hat{Q} = A^T \hat{P}$. A numerically stable LU solver may be used first to invert G , then to invert G^T . The choice of uniform knots leads to good stability, but it is necessary to make certain that the number of control points is smaller than the number of samples by a half. This is essentially a Nyquist frequency argument. If you have as many control points as samples, the B-spline curve can have large oscillations. This is not an issue for us, because our goal is to choose a small number of control points relative to the number of samples in order to reduce our memory requirements.

5.2.2 EVALUATION OF A B-SPLINE CURVE

The B-spline curve control points are computed off-line, so the efficiency of the evaluation of Equation (5.6) is not of concern for the construction. However, the B-spline curve must be evaluated frequently during the program's execution, so it is important to minimize the amount of computation time for each evaluation.

The straightforward method for evaluation of $\mathbf{X}(u)$ in Equation (5.6) is to compute all of $N_{i,d}(u)$ for $0 \leq i \leq n$ using the recursive formulas from Equations (5.7) and (5.8). The pseudocode to compute the basis function values follows. The number n , degree d , and control points $Q[i]$ are assumed to be globally accessible.

```
float N (int i, int j, float u)
{
    if (j > 0)
    {
        c0 = (u - GetKnot(i))/(GetKnot(i+j) - GetKnot(i));
        c1 = (GetKnot(i+j+1) - u)/(GetKnot(i+j+1) - GetKnot(i+1));
        return c0 * N(i,j-1,u) + c1 * N(i+1,j-1,u);
    }
    else // j == 0
    {
        return (GetKnot(i) <= u && u < GetKnot(i+1) ? 1 : 0);
    }
}

Point X (float u)
{
    Point result = ZERO;
    for (i = 0; i <= n; i++)
    {
        result += N(i,d,u) * Q[i];
    }
    return result;
}
```

The knot lookup is

```
float GetKnot (int i)
{
    if (i <= d) return 0;
    if (i >= n+1) return 0;
    return (i-d)/(n+1-d);
}
```

Table 5.1 Recursive dependencies for B-spline basis functions for $n = 5$ and $d = 3$.

	$N_{0,3}$	$N_{1,3}$	$N_{2,3}$	$N_{3,3}$	$N_{4,3}$	$N_{5,3}$													
	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘
	$N_{0,2}$	$N_{1,2}$	$N_{2,2}$	$N_{3,2}$	$N_{4,2}$	$N_{5,2}$	$N_{6,2}$												
	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘
	$N_{0,1}$	$N_{1,1}$	$N_{2,1}$	$N_{3,1}$	$N_{4,1}$	$N_{5,1}$	$N_{6,1}$	$N_{7,1}$											
	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘
	$N_{0,0}$	$N_{1,0}$	$N_{2,0}$	$N_{3,0}$	$N_{4,0}$	$N_{5,0}$	$N_{6,0}$	$N_{7,0}$	$N_{8,0}$										
Knots	0	0	0	[0	$\frac{1}{3}$	$\frac{2}{3}$	1)	1	1	1	1								

This is an inefficient algorithm because local control implies that only a small number of the basis function values are nonzero *and* many of the basis functions are evaluated twice. For example, the value $N_{0,d}(u)$ requires computing $N_{0,d-1}(u)$ and $N_{1,d-1}(u)$. The value $N_{1,d}(u)$ also requires computing $N_{1,d-1}(u)$, as well as $N_{2,d-1}(u)$. The recursive dependencies are illustrated in Table 5.1 for $n = 5$ and $d = 3$.

The rows of knot vectors open with brackets and close with parentheses. These indicate that an evaluation for a specified $u \in [0, 1)$ requires searching for the bounding interval $[u_i, u_{i+1})$ containing u . Only those knots in the bracketed portion need to be searched. The search returns the index of the left endpoint i , where $d \leq i \leq n$. For an open knot vector, the knots corresponding to other indices are included for padding.

To avoid the redundant calculations, you might think to evaluate the table from the bottom up rather than from the top down. In the previous example you would compute $N_{i,0}(u)$ for $0 \leq i \leq 8$ and save these for later access. You would then compute $N_{i,1}(u)$ for $0 \leq i \leq 7$ and look up the values $N_{j,0}(u)$ as needed. The next step is to compute $N_{i,2}(u)$ for $0 \leq i \leq 6$, looking up the values $N_{j,1}(u)$ as needed. Finally, the values $N_{i,3}(u)$ are computed for $0 \leq i \leq 5$, looking up the values $N_{j,2}(u)$ as needed. This is a reasonable modification but still not as efficient as it could be. For a single value of u , only *one* of $N_{i,0}(u)$ is 1; the others are all zero. In the previous example suppose that $u \in [u_4, u_5)$ so that $N_{4,0}(u)$ is 1 and all other $N_{i,0}(u)$ are 0. The only nonzero entries from Table 5.1 are shown as boxed quantities in Table 5.2.

The boxed entries cover a triangular portion of the table. The values on the left diagonal edge and on the right vertical edge are computed first since each value effectively depends only on one previous value, the other value already known to be zero. If $N_{i,0}(u) = 1$, the left diagonal edge is generated by

$$N_{i-j,j}(u) = \frac{u_{i+1} - u}{u_{i+1} - u_{i-j+1}} N_{i-j+1,j-1}(u) \quad (5.15)$$

Table 5.2 Nonzero (boxed) values from Table 5.1 for $N_{4,0}(t) = 1$.

$$\begin{array}{cccccccccccccccc}
N_{0,3} & & \boxed{N_{1,3}} & & \boxed{N_{2,3}} & & \boxed{N_{3,3}} & & \boxed{N_{4,3}} & & N_{5,3} & & & & & & & & \\
\downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & & & & & & & \\
N_{0,2} & & N_{1,2} & & \boxed{N_{2,2}} & & \boxed{N_{3,2}} & & \boxed{N_{4,2}} & & N_{5,2} & & N_{6,2} & & & & & & \\
\downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & & & & & \\
N_{0,1} & & N_{1,1} & & N_{2,1} & & \boxed{N_{3,1}} & & \boxed{N_{4,1}} & & N_{5,1} & & N_{6,1} & & N_{7,1} & & & & \\
\downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & & & \\
N_{0,0} & & N_{1,0} & & N_{2,0} & & N_{3,0} & & \boxed{N_{4,0}} & & N_{5,0} & & N_{6,0} & & N_{7,0} & & N_{8,0} & &
\end{array}$$

and the right vertical edge is generated by

$$N_{i,j}(u) = \frac{u - u_i}{u_{i+j} - u_i} N_{i,j-1}(u) \quad (5.16)$$

both evaluated for $1 \leq j \leq d$. The interior values are computed using the recursive formula, Equation (5.8). The pseudocode for computing the curve point is

```

Point X (float u)
{
    float basis[d+1][n+d+1]; // basis[j][i] = N(i,j)

    // Get indices of the potentially nonzero basis functions.
    int imin, imax;
    GetIndices(u,imin,imax);

    // Evaluate left diagonal and right vertical edges.
    for (j = 1; j <= d; j++)
    {
        c0 = (u - GetKnot(i))/(GetKnot(i+j) - GetKnot(i));
        c1 = (GetKnot(i+1) - u)/(GetKnot(i+1) - GetKnot(i-j+1));
        basis[j][i] = c0 * basis[j-1][i];
        basis[j][i-j] = c1 * basis[j-1][i-j+1];
    }

    // Evaluate interior.
    for (j = 2; j <= d; j++)
    {
        for (k = i - j + 1; k < i; k++)

```

```

    {
        c0 = (u - GetKnot(k)) / (GetKnot(k+j) - GetKnot(k));
        c1 = (GetKnot(k+j+1) - u) / (GetKnot(k+j+1) - GetKnot(k+1));
        basis[j][k] = c0 * basis[j-1][k] + c1 * basis[j-1][k+1];
    }

    Point result = ZERO;
    for (j = imin; j <= imax; j++)
    {
        result += basis[d][j] * Q[j];
    }
    return result;
}

```

The function `GetMinIndex` computes index i from the input parameter u . If the input is outside the interval $[0, 1]$, it is clamped to this interval. The pseudocode is

```

void GetIndices (float& u, int& imin, int& imax) const
{
    if (u <= 0)
    {
        u = 0;
        imin = 0;
    }

    if (u >= 1)
    {
        u = 1;
        imin = n-d;
    }

    imin = floor((n + 1 - d) * u);
    imax = imin + d;
}

```

The pseudocode for `Point X (float u)` is efficient in time, but not space. The pseudocode relies on the existence of a large two-dimensional array called `basis` in which all the values from Table 5.2 may be stored. However, as Table 5.2 indicates, only a small number of storage units are required. In the example, the degree is $d = 3$ and ten storage units are required. The last part of `Point X (float u)` uses local control to restrict the basis function evaluation to those that are nonzero. It also uses only the $d + 1$ basis function values from the top row of the table. With some careful programming, we only need storage for those $d + 1$ values. The relevant entries in the rows of the table below the top row may be computed and temporarily stored.

The order of evaluation is to start with the bottom row (only one entry is nonzero), and then evaluate the entries in the next row up, doing so from left to right. In the example, the computations are as follows. The storage is an array v of $d + 1 = 4$ elements.

```
// N(4,0)
v[3] = 1;

//-----

// N(3,1)
v[2] = ((knot[5] - u)/(knot[5] - knot[4]))*v[3];

// N(4,1)
v[3] = ((u - knot[4])/(knot[5] - knot[4]))*v[3];

//-----

// N(2,2)
v[1] = ((knot[5] - u)/(knot[5] - knot[3]))*v[2];

// N(3,2)
v[2] = ((u - knot[3])/(knot[5] - knot[3]))*v[2] +
        ((knot[6] - u)/(knot[6] - knot[4]))*v[3];

// N(4,2)
v[3] = ((u - knot[4])/(knot[6] - knot[4]))*v[3];

//-----

// N(1,3)
v[0] = ((knot[5] - u)/(knot[5] - knot[2]))*v[1];

// N(2,3)
v[1] = ((u - knot[2])/(knot[5] - knot[2]))*v[1] +
        ((knot[6] - u)/(knot[6] - knot[3]))*v[2];

// N(3,3)
v[2] = ((u - knot[3])/(knot[6] - knot[3]))*v[2] +
        ((knot[7] - u)/(knot[7] - knot[4]))*v[3];

// N(4,3)
v[3] = ((u - knot[4])/(knot[7] - knot[4]))*v[3];
```

Taking advantage of a few patterns in these equations, the general code is

```

Point X (float u)
{
    int imin, imax;
    GetIndices(u,imin,imax);

    v[d] = 1;
    for (int r = d - 1; r >= 0; r--)
    {
        int i0 = imax + 1, i1 = r + imax + 1 - d;
        float knot0 = GetKnot(i0), knot1 = GetKnot(i1);
        float invdenom = 1/(knot0 - knot1);
        float coeff1 = (knot0 - u)*invdenom, coeff0;
        v[r] = coeff1*v[r+1];

        for (int c = r + 1; c < d; c++)
        {
            coeff0 = (t - knot1)*invdenom;
            v[c] *= coeff0;

            knot0 = GetKnot(++i0);
            knot1 = GetKnot(++i1);
            invdenom = 1/(knot0 - knot1);
            coeff1 = (knot0 - u) * invdenom;
            v[c] += coeff1*v[c+1];
        }

        coeff0 = (u - knot1)*invdenom;
        v[d] *= coeff0;
    }

    Point result = ZERO;
    for (int i = 0, j = imin; i <= d; i++)
    {
        result += v[i] * Q[j];
    }
    return result;
}

```

First, we can inline the `GetIndices` function call since the curve evaluation function is the only place it is used. Second, we can eliminate the `GetKnot` function calls by computing the knots once, thus avoiding repeated calls to compute the same knot.

Each knot involves a division, and the coefficients of the v array elements themselves involve divisions. We can reduce the number of divisions by multiplying the coefficients by $n + 1 - d$. Specifically, we can make adjustments like the following. Let $q = n + 1$ for the sake of the discussion; the actual source code uses the same naming convention.

```
q = n + 1;
qmd = q - d;
u' = qmd * u;
knot'[i] = qmd * knot[i]; // 0 <= i < 2*d
(u - knot[i]) / (knot[j] - knot[i]) = (u' - knot'[i]) / (knot'[j] - knot'[i]);
(knot[j] - u) / (knot[j] - knot[i]) = (knot'[j] - u') / (knot'[j] - knot'[i]);
```

The number of distinct knots is $2d$, which can be deduced from the fact that the minimum knot index generated by Equation (5.15) is $i - d + 1$ and the maximum knot index generated by Equation (5.16) is $i + d$. The number of indices is $(i + d) - (i - d + 1) + 1 = 2d$. The curve evaluation becomes

```
Point X (float u)
{
    // q = n + 1
    float qmd = q - d, uprime;
    float uprime;
    int imin, imax;
    if (u <= 0)
    {
        uprime = 0;
        imin = 0;
        imax = d;
    }
    else if (u >= 1)
    {
        uprime = qmd;
        imax = q - 1;
        imin = imax - DEGREE;
    }
    else
    {
        uprime = qmd*u;
        imin = floor(uprime);
        imax = imin + d;
    }
}
```



```

float knotprime[2*d];
for (int i0 = 0, i1 = imax+1-d; i0 < 2*d; i0++, i1++)
{
    if (i1 <= d)
    {
        knotprime[i0] = 0;
    }
    else if (i1 >= q)
    {
        knotprime[i0] = qmd;
    }
    else
    {
        knotprime[i0] = i1 - d;
    }
}

v[d] = 1;
for (int r = d - 1; r >= 0; r--)
{
    int i0 = d, i1 = r;
    float knot0 = knotprime[i0], knot1 = knotprime[i1];
    float invdenom = 1/(knot0 - knot1);
    float coeff1 = (knot0 - u)*invdenom, coeff0;
    v[r] = coeff1*v[r+1];

    for (int c = r+1; c < d; c++)
    {
        coeff0 = (u - knot1)*invdenom;
        v[c] *= coeff0;

        knot0 = knotprime[++i0];
        knot1 = knotprime[++i1];
        invdenom = 1/(knot0 - knot1);
        coeff1 = (knot0 - u) * invdenom;
        v[c] += coeff1*v[c+1];
    }

    coeff0 = (u - knot1)*invdenom;
    v[d] *= coeff0;
}

Point result = ZERO;
for (int i = 0, j = imin; i <= d; i++)

```

```

{
    result += v[i] * Q[j];
}
return result;
}

```

The source code that implements all this is part of the LibFoundation project. In particular, the relevant files are in the Curves folder and have the file names Wm4BSplineFitBasis and Wm4BSplineFit. The B-spline basis function evaluation is encapsulated by the BSplineFitBasis class and depends only on the number of control points and the degree. This allows you to share the evaluators among multiple keyframe sequences. The class BSplineFit encapsulates the construction of the control points from the keyframe samples.

5.2.3 OPTIMIZED EVALUATION FOR DEGREE 3

Cubic B-spline curves are commonly used because they have sufficient smoothness to generate smooth keyframe animations, yet they have low degree, which means you do not spend a lot of cycles for the evaluation. You may very well use the methods described previously to evaluate the curves using a BSplineFitBasis object, but this object does not take advantage of the fact that you are using a fixed degree 3.

The optimizations involve rapid calculation of the knotprime values that occur in the previously mentioned pseudocode, and they vary with $q = 4$, $q = 5$, $q = 6$, and $q \geq 7$. In the following: k_i refers to the knotprime[i] value; u is used for the scaled time (the actual time multiplied by $q - 3$); and v_i refers to the storage array in the basis function evaluator class. The array values are computed in the following order:

$$r = 3: v_3 = 1$$

$$r = 2: v_2 = \frac{k_3 - u}{k_3 - k_2} v_3, \quad v_3 = \frac{u - k_2}{k_3 - k_2} v_3$$

$$r = 1: v_1 = \frac{k_3 - u}{k_3 - k_1} v_2, \quad v_2 = \frac{u - k_1}{k_3 - k_1} v_2 + \frac{k_4 - u}{k_4 - k_2} v_3, \quad v_3 = \frac{u - k_2}{k_4 - k_2} v_3$$

$$r = 0: v_0 = \frac{k_3 - u}{k_3 - k_0} v_1, \quad v_1 = \frac{u - k_0}{k_3 - k_0} v_1 + \frac{k_4 - u}{k_4 - k_1} v_2,$$

$$v_2 = \frac{u - k_1}{k_4 - k_1} v_2 + \frac{k_5 - u}{k_5 - k_2} v_3, \quad v_3 = \frac{u - k_2}{k_5 - k_2} v_3$$

Consider the case $q = 4$. The indices for the curve evaluation are always $i_{\min} = 0$ and $i_{\max} = 3$. The knot values are necessarily

$$k_0 = 0, \quad k_1 = 0, \quad k_2 = 0, \quad k_3 = 1, \quad k_4 = 1, \quad k_5 = 1$$

The v values are computed as

$$r = 3: v_3 = 1$$

$$r = 2: v_2 = (1 - u)v_3, \quad v_3 = uv_3$$

$$r = 1: v_1 = (1 - u)v_2, \quad v_2 = uv_2 + (1 - u)v_3, \quad v_3 = uv_3$$

$$r = 0: v_0 = (1 - u)v_1, \quad v_1 = uv_1 + (1 - u)v_2, \quad v_2 = uv_2 + (1 - u)v_3, \quad v_3 = uv_3$$

Replacing each row into the one following it, and factoring out common expressions,

```
// q = 4
one_m_u = 1 - u;
u_sqr = u * u;
one_m_usqr = one_m_u * one_m_u;
v[0] = one_m_u * one_m_usqr;
v[1] = 3 * u * one_m_usqr;
v[2] = 3 * u_sqr * one_m_u;
v[3] = u * u_sqr;
```

Consider the case $q = 5$. The maximum index for the curve evaluation is $i_{\max} \in \{3, 4\}$ and $i_{\min} = i_{\max} - 3$. The knot values are

$$k_0 = 0, \quad k_1 = 0, \quad k_2 = i - 3, \quad k_3 = i - 2, \quad k_4 = 2, \quad k_5 = 2$$

for $i \in \{3, 4\}$. For $i = 3$, the v values are computed as

$$r = 3: v_3 = 1$$

$$r = 2: v_2 = (1 - u)v_3, \quad v_3 = uv_3$$

$$r = 1: v_1 = (1 - u)v_2, \quad v_2 = uv_2 + (2 - u)v_3, \quad v_3 = (u/2)v_3$$

$$r = 0: v_0 = (1 - u)v_1, \quad v_1 = uv_1 + ((2 - u)/2)v_2,$$

$$v_2 = (u/2)v_2 + ((2 - u)/2)v_3, \quad v_3 = (u/2)v_3$$

Replacing each row into the one following it, and factoring out common expressions,

```
// q = 5, i = 3
one_m_u = 1 - u;
two_m_u = 2 - u;
half_u = 0.5 * u;
one_m_usqr = one_m_u * one_m_u;
expr = 0.5 * (u * one_m_u + two_m_u * half_u);
halfu_sqr = half_u * half_u;
```

```

v[0] = one_m_u * one_m_usqr;
v[1] = u * one_m_usqr + two_m_u * expr;
v[2] = u * expr + two_m_u * halfu_sqr;
v[3] = u * halfu_sqr;

```

For $i = 4$, the v values are computed as

$$\begin{aligned}
r = 3: v_3 &= 1 \\
r = 2: v_2 &= \frac{k_3 - u}{k_3 - k_2} v_3, \quad v_3 = \frac{u - k_2}{k_3 - k_2} v_3 \\
r = 1: v_1 &= \frac{k_3 - u}{k_3 - k_1} v_2, \quad v_2 = \frac{u - k_1}{k_3 - k_1} v_2 + \frac{k_4 - u}{k_4 - k_2} v_3, \quad v_3 = \frac{u - k_2}{k_4 - k_2} v_3 \\
r = 0: v_0 &= \frac{k_3 - u}{k_3 - k_0} v_1, \quad v_1 = \frac{u - k_0}{k_3 - k_0} v_1 + \frac{k_4 - u}{k_4 - k_1} v_2, \\
v_2 &= \frac{u - k_1}{k_4 - k_1} v_2 + \frac{k_5 - u}{k_5 - k_2} v_3, \quad v_3 = \frac{u - k_2}{k_5 - k_2} v_3
\end{aligned}$$

Replacing each row into the one following it, and factoring out common expressions,

```

// q = 5, i = 4
u_m_one = u - 1;
two_m_u = 2 - u;
half_u = 0.5 * u;
umone_sqr = u_m_one * u_m_one;
one_m_halfu = 1 - half_u;
onemhalfu_sqr = one_m_halfu * one_m_halfu;
expr = one_m_halfu * (half_u + u_m_one);
v[0] = two_m_u * onemhalfu_sqr;
v[1] = u * onemhalfu_sqr + two_m_u * expr;
v[2] = u * expr + two_m_u * umone_sqr;
v[3] = u_m_one * umone_sqr;

```

Consider the case $q = 6$. The maximum index for the curve evaluation is $i_{\max} \in \{3, 4, 5\}$ and $i_{\min} = i_{\max} - 3$. The knot values are

$$\begin{aligned}
k_0 = 0, k_1 &= \begin{cases} 0, & 3 \leq i \leq 4 \\ 1, & i = 5 \end{cases}, k_2 = i - 3, k_3 = i - 2, \\
k_4 &= \begin{cases} 2, & i = 3 \\ 3, & 4 \leq i \leq 5 \end{cases}, k_5 = 3
\end{aligned}$$

For $i = 3$, the v values are computed as

$$\begin{aligned}
r = 3: v_3 &= 1 \\
r = 2: v_2 &= (1 - u)v_3, \quad v_3 = uv_3 \\
r = 1: v_1 &= (1 - u)v_2, \quad v_2 = (u - 1)v_2 + (2 - u)v_3, \quad v_3 = (u/2)v_3 \\
r = 0: v_0 &= (1 - u)v_1, \quad v_1 = uv_1 + ((2 - u)/2)v_2, \\
&v_2 = (u/2)v_2 + ((3 - u)/3)v_3, \quad v_3 = (u/3)v_3
\end{aligned}$$

Replacing each row into the one following it, and factoring out common expressions,

```
// q = 6, i = 3
one_m_u = 1 - u;
two_m_u = 2 - u;
three_m_u = 3 - u;
half_u = 0.5 * u;
onemu_sqr = one_m_u * one_m_u;
expr0 = 0.5 * (u * one_m_u + two_m_u * half_u);
expr1 = u * half_u / 3;
v[0] = one_m_u * onemu_sqr;
v[1] = u * onemu_sqr + two_m_u * expr0;
v[2] = u * expr0 + three_m_u * expr1;
v[3] = u * expr1;
```

For $i = 4$, the v values are computed as

$$\begin{aligned}
r = 3: v_3 &= 1 \\
r = 2: v_2 &= (2 - u)v_3, \quad v_3 = (u - 1)v_3 \\
r = 1: v_1 &= ((2 - u)/2)v_2, \quad v_2 = (u/2)v_2 + ((3 - u)/2)v_3, \quad v_3 = ((u - 1)/2)v_3 \\
r = 0: v_0 &= ((2 - u)/2)v_1, \quad v_1 = (u/2)v_1 + ((3 - u)/3)v_2, \\
&v_2 = (u/3)v_2 + ((3 - u)/2)v_3, \quad v_3 = ((u - 1)/2)v_3
\end{aligned}$$

Replacing each row into the one following it, and factoring out common expressions,

```
// q = 6, i = 4
u_m_one = u - 1;
two_m_u = 2 - u;
three_m_u = 3 - u;
half_u = 0.5 * u;
one_m_halfu = 1 - half_u;
half_umone = 0.5 * u_m_one;
onemhalfu_sqr = one_m_halfu * one_m_halfu;
expr = (u * one_m_halfu + three_m_u * half_umone) / 3;
halfumone_sqr = half_umone * half_umone;
```

```

v[0] = two_m_u * onemhalfu_sqr;
v[1] = u * onemhalfu_sqr + three_m_u * expr;
v[2] = u * expr + three_m_u * halfumone_sqr;
v[3] = u_m_one * halfumone_sqr;

```

For $i = 5$, the v values are computed as

$$\begin{aligned}
r = 3: v_3 &= 1 \\
r = 2: v_2 &= (3 - u)v_3, \quad v_3 = (u - 2)v_3 \\
r = 1: v_1 &= ((3 - u)/2)v_2, \quad v_2 = ((u - 1)/2)v_2 + (3 - u)v_3, \quad v_3 = (u - 2)v_3 \\
r = 0: v_0 &= ((3 - u)/3)v_1, \quad v_1 = (u/3)v_1 + ((3 - u)/2)v_2, \\
&\quad v_2 = ((u - 1)/2)v_2 + (3 - u)v_3, \quad v_3 = (u - 2)v_3
\end{aligned}$$

Replacing each row into the one following it, and factoring out common expressions,

```

// q = 6, i = 5
u_m_one = u - 1;
u_m_two = u - 2;
three_m_u = 3 - u;
half_three_m_u = 0.5 * three_m_u;
umtwo_sqr = u_m_two * u_m_two;
expr0 = three_m_u * half_three_m_u / 3;
expr1 = 0.5 * (u_m_one * half_three_m_u + three_m_u * u_m_two);
v[0] = three_m_u * expr0;
v[1] = u * expr0 + three_m_u * expr1;
v[2] = u_m_one * expr1 + three_m_u * umtwo_sqr;
v[3] = u_m_two * umtwo_sqr;

```

Finally, consider the case $q \geq 7$. This is the expected case—the animation sequences have a lot of data. Two of the knot values are $k_2 = i - 3$ and $k_3 = i - 2$, where $i_{\min} \leq i \leq i_{\max}$. The difference in consecutive knots is either 0 or 1. The formal definitions are listed next, where $q = n + 1 \geq 7 = 2d + 1$.

$$\begin{aligned}
k_0 &= \left\{ \begin{array}{ll} 0, & 3 \leq i \leq 5 \\ i - 5, & 6 \leq i \leq q - 1 \end{array} \right\}, & k_1 &= \left\{ \begin{array}{ll} 0, & 3 \leq i \leq 4 \\ i - 4, & i = 5 \leq i \leq q - 1 \end{array} \right\}, \\
k_2 &= \{ i - 3, \quad 3 \leq i \leq q - 1 \}, & k_3 &= \{ i - 2, \quad 3 \leq i \leq q - 1 \}, \\
k_4 &= \left\{ \begin{array}{ll} i - 1, & i = 3 \leq i \leq q - 3 \\ q - 3, & q - 2 \leq i \leq q - 1 \end{array} \right\}, & k_5 &= \left\{ \begin{array}{ll} i, & i = 3 \leq i \leq q - 4 \\ q - 3, & q - 3 \leq i \leq q - 1 \end{array} \right\}
\end{aligned}$$

The algorithm requires computing the reciprocals of $k_3 - k_2, k_3 - k_1, k_4 - k_2, k_3 - k_0, k_4 - k_1$, and $k_5 - k_2$. It also requires computing $u - k_0, u - k_1, u - k_2, k_3 - u, k_4 - u,$

and $k_5 - u$. The general formulas for the v array values apply and use the previously mentioned 12 precomputed numbers. Replacing each row into the one following it, and factoring out common expressions,

```

oneThird = 1.0 / 3.0;
qm2 = qm3 + 1;
qm1 = qm2 + 1;

g0 = (i > 5 ? i-5 : 0.0);
g1 = (i > 4 ? i-4 : 0.0);
g2 = i-3;
g3 = i-2;
g4 = (i < qm2 ? i-1 : qm3);
g5 = (i < qm3 ? i : qm3);

inv_g3_m_g1 = (i == 3 ? 1.0 : 0.5);
inv_g4_m_g2 = (i == qm1 ? 1.0 : 0.5);
inv_g3_m_g0 = (i == 3 ? 1.0 : (i == 4 ? 0.5 : oneThird));
inv_g4_m_g1 = (i == 3 || i == qm1 ? 0.5 : oneThird);
inv_g5_m_g2 = (i == qm1 ? 1.0 : (i == qm2 ? 0.5 : oneThird));

u_m_g0 = u - g0;
u_m_g1 = u - g1;
u_m_g2 = u - g2;
g3_m_u = g3 - u;
g4_m_u = g4 - u;
g5_m_u = g5 - u;

expr0 = g3_m_u * inv_g3_m_g1;
expr1 = u_m_g2 * inv_g4_m_g2;
expr2 = inv_g3_m_g0 * g3_m_u * expr0;
expr3 = inv_g4_m_g1 * (u_m_g1 * expr0 + g4_m_u * expr1);
expr4 = inv_g5_m_g2 * u_m_g2 * expr1;

v[0] = g3_m_u * expr2;
v[1] = u_m_g0 * expr2 + g4_m_u * expr3;
v[2] = u_m_g1 * expr3 + g5_m_u * expr4;
v[3] = u_m_g2 * expr4;

```

Profiling of the generic code versus the optimized code on my computers showed about a 20% speedup for the B-spline evaluations.

EXERCISE 5.1

Repeat the construction of this section for B-spline curves of degree 2. The idea is to produce an optimized evaluation of such B-spline curves. Compare execution times between the optimized version and the generic version to see what the speedup is. ■

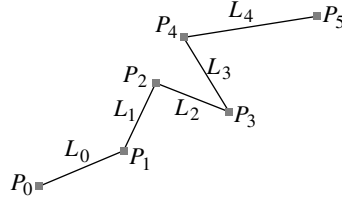


Figure 5.1 A general linearly linked manipulator.

5.3 INVERSE KINEMATICS

Kinematics is the study of motion without consideration of mass or forces. We will illustrate the ideas first in two dimensions. Given a planar polyline consisting of a sequence of line segments (or bones, so to speak), with each segment starting at $\mathbf{P}_i$, having unit-length direction $\mathbf{U}_i$, and length L_i for $0 \leq i < n$, and with the last segment terminating at $\mathbf{P}_n$, the *forward kinematics* problem is to compute $\mathbf{P}_n$ in terms of the known direction vectors and lengths. The structure is called a *manipulator*, and the final point is called an *end effector*. Figure 5.1 illustrates the general setting.

The final point of each segment is related to its starting point by $\mathbf{P}_{i+1} = \mathbf{P}_i + L_i \mathbf{U}_i$ for $0 \leq i < n$. Summing over all i and canceling the common terms leads to the end effector formula

$$\mathbf{P}_n = \mathbf{P}_0 + \sum_{i=0}^{n-1} L_i \mathbf{U}_i$$

Each direction vector can be viewed as an incremental rotation of the previous direction,

$$\mathbf{U}_i = \left(\cos \left(\sum_{j=0}^i \theta_j \right), \sin \left(\sum_{j=0}^i \theta_j \right) \right)$$

The angles θ_j are called the *joint angles* of the manipulator. Using notation $\boldsymbol{\theta} = (\theta_0, \dots, \theta_{n-1})$, the end effector can be written as a function:

$$\mathbf{P}_n(\boldsymbol{\theta}) = \mathbf{P}_0 + \sum_{i=0}^{n-1} L_i \left(\cos \left(\sum_{j=0}^i \theta_j \right), \sin \left(\sum_{j=0}^i \theta_j \right) \right) \quad (5.17)$$

The *inverse kinematics* problem is to select the position $\mathbf{G}$ for the end effector and determine joint angles $\boldsymbol{\theta}$ so that $\mathbf{P}_n(\boldsymbol{\theta}) = \mathbf{G}$. The point $\mathbf{G}$ is called the *goal* and might not always be attainable. This is definitely the case when the distance of the

goal to the initial point of the manipulator is larger than the sum of the lengths of the segments. Even if the goal is attainable, there may be multiple solutions. Thus, the inverse problem is generally ill-posed.

Obtaining a closed-form representation of the joint angles in terms of $\mathbf{G}$, $\mathbf{P}_0$, and the L_i is a hard problem. To show the complexity, consider the case of two segments. The equation to be inverted is $\mathbf{G} = \mathbf{P}_0 + L_0\mathbf{U}_0 + L_1\mathbf{U}_1$, where $\mathbf{U}_0 = (\cos \theta_0, \sin \theta_0)$ and $\mathbf{U}_1 = (\cos(\theta_0 + \theta_1), \sin(\theta_0 + \theta_1))$. Define $(a, b)^\perp = (-b, a)$. Using the double-angle identities from trigonometry, $\mathbf{U}_1 = (\cos \theta_1)\mathbf{U}_0 + (\sin \theta_1)\mathbf{U}_0^\perp$. The equation to be solved is therefore

$$\mathbf{G} = \mathbf{P}_0 + (L_0 + L_1 \cos \theta_1)\mathbf{U}_0 + (L_1 \sin \theta_1)\mathbf{U}_0^\perp = \mathbf{P}_0 + R_0\mathbf{V}_1$$

where $R_0 = [\mathbf{U}_0 \mid \mathbf{U}_0^\perp]$ is a rotation matrix and $\mathbf{V}_1 = (L_0 + L_1 \cos \theta_1, L_1 \sin \theta_1)$.

Note that $\mathbf{G} - \mathbf{P}_0 = R_0\mathbf{V}_1$, so the difference between the goal and the initial point is just a rotation of $\mathbf{V}_1$. Since rotation preserves length,

$$|\mathbf{G} - \mathbf{P}_0|^2 = |\mathbf{V}_1|^2 = L_0^2 + L_1^2 + 2L_0L_1 \cos \theta_1$$

in which case

$$\cos \theta_1 = \frac{|\mathbf{G} - \mathbf{P}_0|^2 - L_0^2 - L_1^2}{2L_0L_1}$$

There are two possible choices for the sine, $\sin \theta_1 = \pm\sqrt{1 - \cos^2 \theta_1}$. The other angle is determined by

$$|\mathbf{V}_1|^2 \cos \theta_0 = \mathbf{V}_1 \cdot R_0\mathbf{V}_1 = \mathbf{V}_1 \cdot (\mathbf{G} - \mathbf{P}_0)$$

and

$$|\mathbf{V}_1|^2 \sin \theta_0 = \mathbf{V}_1^\perp \cdot R_0\mathbf{V}_1 = \mathbf{V}_1^\perp \cdot (\mathbf{G} - \mathbf{P}_0)$$

As long as $|\mathbf{G} - \mathbf{P}_0| < L_0 + L_1$ there are two solutions, as indicated by the sign choice for $\sin \theta_1$. This is clear geometrically since one manipulator configuration is obtained from the other by reflection through the line containing the initial point and goal.

The inverse kinematics problem can be complicated even more by allowing quite a few variations. The preceding example was two-dimensional. The real problems are three-dimensional. Each joint has six degrees of freedom, three for position and three for orientation. The degrees of freedom can be additionally constrained within their parameter space. The typical constraints are to restrict rotation about a single axis, in which case the joint is called a *revolute joint*, and/or to restrict translation along the direction of the previous segment, in which case the joint is called a *prismatic joint*. Moreover, within the restrictions the parameters might themselves be constrained. At a revolute joint, the angle of rotation might be limited to a subset of $[0, 2\pi]$. At a prismatic joint, the translation might be constrained to be a small interval

$[-\epsilon, \epsilon]$. Finally, manipulators can be trees of segments rather than lists of segments. The leaves of the trees represent the end effectors, so there are multiple goals that can be specified. Some attempt must be made to simultaneously satisfy the goals, or at least to get close to the goals. Yet another variation is to specify goals that are lines or planes. The end effector is considered to be in its best position when the distance from the end effector to the goal is minimized. At any rate, closed-form solutions are usually not possible—or even desirable—because they involve evaluation of trigonometric functions. Numerical methods are a better choice for attempting to find solutions.

One of the best discussions for inverse kinematics is [Wel93]. A well-written summary of the topic is [Lan98].

5.3.1 NUMERICAL SOLUTION BY JACOBIAN METHODS

Consider a manipulator that is a polyline with a single end effector. Let the end effector be written as $\mathbf{P} = \mathbf{F}(\boldsymbol{\theta})$, a function of the joint angles $\boldsymbol{\theta}$. The derivative of the end effector position with respect to each joint parameter θ_i can be used to determine an incremental step in joint space that will (hopefully) move the end effector closer to the goal. If the position of the end effector is thought of as moving, hence a function of time t , the derivatives are

$$\frac{d\mathbf{P}}{dt} = D\mathbf{F} \frac{d\boldsymbol{\theta}}{dt}$$

where $D\mathbf{F}$ is the *Jacobian* of $\mathbf{F}$, the matrix of first-order partial derivatives,

$$D\mathbf{F} = \begin{bmatrix} \frac{\partial F_i}{\partial \theta_j} \end{bmatrix}$$

where F_i is the i th component of $\mathbf{F}$ and θ_j is the j th component of $\boldsymbol{\theta}$. The time-derivative equation relates the end effector velocity to the joint velocities.

If $\mathbf{G}$ is the goal and if $d\mathbf{P}/dt$ is replaced by $\mathbf{G} - \mathbf{P} = \mathbf{G} - \mathbf{F}(\boldsymbol{\theta})$ as an approximation, then the numerical method is to use $\mathbf{G} - \mathbf{F}(\boldsymbol{\theta}) = D\mathbf{F}(\boldsymbol{\theta})d\boldsymbol{\theta}/dt$ to update $\boldsymbol{\theta}$ from its current value. The Jacobian matrix is (usually) not square, so its inverse is not defined. However, given a nonsquare matrix M , its pseudoinverse is defined to be $M^+ = M^T(MM^T)^{-1}$, where $M^+M = I$, the identity matrix. Applying the pseudoinverse of the Jacobian yields

$$\frac{d\boldsymbol{\theta}}{dt} = D\mathbf{F}^+(\boldsymbol{\theta})(\mathbf{G} - \mathbf{F}(\boldsymbol{\theta}))$$

Given a current value of $\boldsymbol{\theta}$, this equation allows an update by using a forward difference operator to approximate the time derivative of the joint angles. The scheme is applied iteratively until some stopping criterion is met.

This approach is not always the best one since computing the pseudoinverse is expensive (a square matrix inverse is required). Moreover, sometimes the Jacobian is singular on its domain or is ill-conditioned, so numerical problems arise in the inversion. A different approach is to avoid the inversion and apply the transpose of the Jacobian to obtain

$$\boldsymbol{\tau} = D\mathbf{F}^T \frac{d\mathbf{P}}{dt} = \left(D\mathbf{F}^T D\mathbf{F} \right) \frac{d\boldsymbol{\theta}}{dt}$$

The value $\boldsymbol{\tau}$ measures the amount of torque at the joints induced by a force $d\mathbf{P}/dt$. If the torque is computed for the current joint angles using $\mathbf{G} - \mathbf{F}(\boldsymbol{\theta})$ instead of $d\mathbf{P}/dt$, the unknown vector is $\mathbf{x} = d\boldsymbol{\theta}/dt$. The displayed equation is of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$ and might not always have a solution. However, minimization methods can be applied to $\mathbf{e}(\mathbf{x}) = |\mathbf{A}\mathbf{x} - \mathbf{b}|^2$ to obtain a solution $\mathbf{x}$. Again using a forward difference approximation, this allows an update of the current joint angles. For more on Jacobian methods, see [SS87, DSS88, NN90].

5.3.2 NUMERICAL SOLUTION BY NONLINEAR OPTIMIZATION

This is a general approach that can take advantage of already existing algorithms for optimization. The idea is to minimize the squared error $E(\boldsymbol{\theta}) = |\mathbf{G} - \mathbf{F}(\boldsymbol{\theta})|^2$ with respect to $\boldsymbol{\theta}$. While the goal indicator here is a point, the same type of error function applies for goals that are lines or planes. Secondary goals are easily incorporated into the error function. The results using general optimization are generally good, but the algorithm tends to be expensive. For more on nonlinear optimization methods, see [PZB90, ZB94].

5.3.3 NUMERICAL SOLUTION BY CYCLIC COORDINATE DESCENT

The *cyclic coordinate descent* approach was introduced in [WC91]. The idea is conceptually simple, and the algorithm is fast. The joints of the manipulator are optimized one at a time, and several passes are made over the manipulator to (hopefully) arrive at the global minimum of $|\mathbf{P} - \mathbf{F}(\boldsymbol{\theta})|$. As with most minimization schemes, local minima can attract the iterates. In terms of manipulators, this can happen if the polyline has a kink in it that cannot be undone by successive iterations. For the purpose of animation, secondary goals or restrictions on joint angles can be added to avoid such behavior.

List Manipulator with One End Effector

Consider a list manipulator with initial point $\mathbf{I}$ and lengths L_i for $0 \leq i < n$. The update at a single joint is discussed for goals that are points, lines, or planes. The joint can be revolute or prismatic.

Rotate to Point

Let the goal be $\mathbf{G}$. If the rotation is unconstrained, then the end effector position $\mathbf{E}$ is chosen so that it lies on the line containing $\mathbf{I}$ and $\mathbf{G}$. The position is $\mathbf{E} = \mathbf{I} + t(\mathbf{G} - \mathbf{I})$, where $t = (\mathbf{G} - \mathbf{I}) \cdot (\mathbf{E} - \mathbf{I}) / |\mathbf{G} - \mathbf{I}|^2$.

If the rotation is constrained to the plane $\mathbf{N} \cdot (\mathbf{X} - \mathbf{I}) = 0$, where $\mathbf{N}$ is unit length, then the end effector position is chosen so that it lies on the line containing $\mathbf{I}$ and the projection of $\mathbf{G}$ onto the plane. The previous case applies using the projection. The projection is $\mathbf{H} = \mathbf{G} - [\mathbf{N} \cdot (\mathbf{G} - \mathbf{I})]\mathbf{N}$.

Rotate to Line

Let the goal be $\mathbf{G}(s) = \mathbf{G}_0 + s\mathbf{G}_1$ for $s \in \mathbb{R}$ and where $|\mathbf{G}_1| = 1$. If the rotation is unconstrained, there are two cases to consider. The closest point on the line to $\mathbf{I}$ is

$$\mathbf{J} = \mathbf{G}_0 - [\mathbf{G}_1 \cdot (\mathbf{G}_0 - \mathbf{I})]\mathbf{G}_1$$

and the distance from $\mathbf{I}$ to the line is $D = |\mathbf{J} - \mathbf{I}|$. If $D \geq |\mathbf{E} - \mathbf{I}|$, then the end effector is chosen so that it lies on the line containing $\mathbf{I}$ and $\mathbf{J}$. The position is $\mathbf{E} = \mathbf{I} + t(\mathbf{J} - \mathbf{I})$, where $t = (\mathbf{J} - \mathbf{I}) \cdot (\mathbf{E} - \mathbf{I}) / |\mathbf{J} - \mathbf{I}|^2$. If $D < |\mathbf{E} - \mathbf{I}|$, then there are two solutions that lie on the line itself, $\mathbf{J} \pm R\mathbf{G}_1$. The quantity R is determined from the Pythagorean theorem applied to the right triangle containing vertices $\mathbf{I}$ and $\mathbf{J}$ and having hypotenuse $|\mathbf{E} - \mathbf{I}|$. Thus, $R^2 = |\mathbf{E} - \mathbf{I}|^2 - |\mathbf{J} - \mathbf{I}|^2$. In an iterative scheme, the end effector will be updated to the nearest of the two points.

If the rotation is constrained to the plane $\mathbf{N} \cdot (\mathbf{X} - \mathbf{I}) = 0$, where $\mathbf{N}$ is unit length, then the line is projected onto that plane and the previous case applies using the projected line. The projected line is $\mathbf{H}_0 + s\mathbf{H}_1$, where $\mathbf{H}_0 = \mathbf{G}_0 - [\mathbf{N} \cdot (\mathbf{G}_0 - \mathbf{I})]\mathbf{N}$ and $\mathbf{H}_1 = \mathbf{G}_1 - (\mathbf{N} \cdot \mathbf{G}_1)\mathbf{N}$.

Rotate to Plane

Let the goal be $\mathbf{M} \cdot \mathbf{X} = c$, where $\mathbf{M}$ is unit length. If the rotation is unconstrained, then there are two cases to consider. The closest point on the plane to $\mathbf{I}$ is

$$\mathbf{J} = \mathbf{I} - (\mathbf{M} \cdot \mathbf{I} - c)\mathbf{M}$$

and the distance from $\mathbf{I}$ to the plane is $D = |\mathbf{J} - \mathbf{I}| = |\mathbf{M} \cdot \mathbf{I} - c|$. If $D \geq |\mathbf{E} - \mathbf{I}|$, then the end effector is chosen so that it lies on the line containing $\mathbf{I}$ and $\mathbf{J}$. The position is $\mathbf{E} = \mathbf{I} + t(\mathbf{J} - \mathbf{I})$, where $t = (\mathbf{J} - \mathbf{I}) \cdot (\mathbf{E} - \mathbf{I}) / |\mathbf{J} - \mathbf{I}|^2$. If $D < |\mathbf{E} - \mathbf{I}|$, then there are infinitely many solutions that lie on a circle in the goal plane that is centered at $\mathbf{J}$ and has radius R . The radius is determined in a way similar to when the goal is a line, $R^2 = |\mathbf{E} - \mathbf{I}|^2 - |\mathbf{J} - \mathbf{I}|^2$. In an iterative scheme, the end effector will be updated to the nearest of the circle points. Finding the nearest point on a circle in three dimensions is discussed in Section 14.13.4.

Let the rotation be constrained to the plane $\mathbf{N} \cdot (\mathbf{X} - \mathbf{I}) = 0$, where $\mathbf{N}$ is unit length. If $\mathbf{N}$ and $\mathbf{M}$ are parallel, then the circle of possible end effector positions is parallel to the goal plane, in which case no motion is necessary. If the two plane normals are not parallel, then the circle of positions is $\mathbf{F}(\theta) = \mathbf{I} + (\cos \theta)(\mathbf{E} - \mathbf{I}) + (\sin \theta)\mathbf{N} \times (\mathbf{E} - \mathbf{I})$, where $\mathbf{E}$ is the current end effector position. The signed distance from any circle point to the plane is

$$s(\theta) = \mathbf{M} \cdot \mathbf{F}(\theta) - c = (\mathbf{M} \cdot \mathbf{I} - c) + \mu_0 \cos \theta + \mu_1 \sin \theta$$

where $\mu_0 = \mathbf{M} \cdot (\mathbf{E} - \mathbf{I})$ and $\mu_1 = \mathbf{M} \cdot \mathbf{N} \times (\mathbf{E} - \mathbf{I})$. In the case under consideration, the circle is not parallel to the plane, so $\mu_0^2 + \mu_1^2 = |\mathbf{M} - (\mathbf{M} \cdot \mathbf{N})\mathbf{N}|^2 \neq 0$. The range of $s(\theta)$ is $[s_{\min}, s_{\max}]$, where

$$s_{\min} = \mathbf{M} \cdot \mathbf{I} - c - \sqrt{\mu_0^2 + \mu_1^2}$$

and

$$s_{\max} = \mathbf{M} \cdot \mathbf{I} - c + \sqrt{\mu_0^2 + \mu_1^2}$$

If $0 \in [s_{\min}, s_{\max}]$, then the circle intersects the goal plane for two values of θ . Define $\lambda = c - \mathbf{M} \cdot \mathbf{I}$, $x_0 = \cos \theta$, $x_1 = \sin \theta$, and set $s(\theta) = 0$ to obtain $\mu_0 x_0^2 + \mu_1^2 x_1^2 = \lambda$ and $x_0^2 + x_1^2 = 1$. These form a polynomial system—one linear and one quadratic equation—that can be solved by resultants (see Section 16.1.2). The resultant is $r(x_0) = (\mu_0^2 + \mu_1^2)x_0^2 - 2\lambda\mu_0 x_0 + \lambda^2 - \mu_1^2 = 0$. The solutions are

$$\cos \theta = \frac{\lambda\mu_0 \pm \mu_1 \sqrt{\mu_0^2 + \mu_1^2 - \lambda^2}}{\mu_0^2 + \mu_1^2}$$

If $0 \notin [s_{\min}, s_{\max}]$, then observe

$$\mu_0 \cos \theta + \mu_1 \sin \theta = \sqrt{\mu_0^2 + \mu_1^2} \cos(\theta - \phi)$$

where $\tan \phi = \mu_1/\mu_0$. If $s_{\min} > 0$, then the closest point occurs when $\theta - \phi = \pi$. If $s_{\max} < 0$, then the closest point occurs when $\theta - \phi = 0$.

Slide to Point

Sliding refers to the linear motion of an endpoint of a segment in the manipulator. If $\mathbf{I}$ is the initial point of a segment and the direction of the segment is the unit-length vector $\mathbf{U}$, unconstrained motion allows the final point to be $\mathbf{F} = \mathbf{I} + t\mathbf{U}$ for $t \in [0, \infty)$. Constrained motion requires $t \in [t_{\min}, t_{\max}]$, where the interval is application-specific. The path traveled by the end effector position relative to the sliding motion is $\mathbf{E} + t\mathbf{U}$ for t in the appropriate interval.

Let the goal be $\mathbf{G}$. The projection onto the ray containing $\mathbf{E}$ with direction $\mathbf{U}$ is $\mathbf{H} = \mathbf{E} + T\mathbf{U}$, where $T = \mathbf{U} \cdot (\mathbf{G} - \mathbf{E})$. If the sliding is unconstrained, then the end effector position $\mathbf{E}$ is updated to $\mathbf{F} = \mathbf{E} + \max\{0, T\}\mathbf{U}$. If the sliding is constrained,

then the end effector position is updated to $\mathbf{F} = \mathbf{E} + \text{clamp}\{T, t_{\min}, t_{\max}\}$, where

$$\text{clamp}\{T, t_{\min}, t_{\max}\} = \begin{cases} t_{\max}, & T > t_{\max} \\ T, & T \in [t_{\min}, t_{\max}] \\ t_{\min}, & T < t_{\min} \end{cases}$$

Slide to Line

Let the goal be $\mathbf{G}(s) = \mathbf{G}_0 + s\mathbf{G}_1$ for $s \in \mathbb{R}$ and where $|\mathbf{G}_1| = 1$. If the goal line is parallel to the direction of sliding, $\mathbf{U} \times \mathbf{G}_1 = \mathbf{0}$, then no updating of $\mathbf{E}$ is necessary. Otherwise, the lines are not parallel and the closest point on the ray containing $\mathbf{E}$ with direction $\mathbf{U}$ is $\mathbf{H} = \mathbf{E} + T\mathbf{U}$, where $T \geq 0$ (see Section 14.1.2). The update of $\mathbf{E}$ in the previous subsection on sliding to a point can now be applied using $\mathbf{H}$.

Slide to Plane

Let the goal be $\mathbf{M} \cdot \mathbf{X} = c$, where $\mathbf{M}$ is unit length. If the goal plane is parallel to the direction of sliding, $\mathbf{M} \cdot \mathbf{U} = 0$, then no updating of $\mathbf{E}$ is necessary. Otherwise, let $\mathbf{H}$ on the ray containing $\mathbf{E}$ with direction $\mathbf{U}$ be the closest point to the plane. The update of $\mathbf{E}$ is the same as for that shown in sliding to a point.

List Manipulator with Multiple End Effectors

Consider the example of a two-segment manipulator whose initial point corresponds to a shoulder, whose midpoint corresponds to an elbow, and whose final point corresponds to a hand. If a point goal is specified for the hand end effector, it is possible that obtaining the goal requires bending the elbow joint in an unnatural way. The algorithm for rotation to a point described earlier could be modified to include a restriction on the angles for the elbow to prevent the unnatural bending. Alternatively, the elbow itself can be tagged as an end effector, and a secondary goal can be specified that affects the elbow location.

Let $\mathbf{S}$ be the shoulder location, $\mathbf{E}$ be the elbow location, and $\mathbf{H}$ be the hand location. Let $\mathbf{G}_H$ be the goal for the hand and let $\mathbf{G}_E$ be the goal for the elbow. Consider adjusting the joint angles at the shoulder. If only the hand goal is required, the rotate-to-point algorithm minimized the distance from $\mathbf{H}$ to $\mathbf{G}_H$. To handle multiple end effectors, the minimization algorithm applies to a weighted sum of squared distances, $D = w_H|\mathbf{H} - \mathbf{G}_H|^2 + w_E|\mathbf{E} - \mathbf{G}_E|^2$, where the weights are application-specific. The number of independent parameters for D depends on whether or not the shoulder joint is unconstrained.

To illustrate how the minimization applies, consider a constrained rotation with the plane $\mathbf{N} \cdot (\mathbf{X} - \mathbf{S}) = 0$. The circle spanned by $\mathbf{H}$ is

$$\mathbf{h}(\theta) = \mathbf{S} + (\cos \theta)(\mathbf{H} - \mathbf{S}) + (\sin \theta)\mathbf{N} \times (\mathbf{H} - \mathbf{S})$$

and the circle spanned by $\mathbf{E}$ is

$$\mathbf{e}(\theta) = \mathbf{S} + (\cos \theta)(\mathbf{E} - \mathbf{S}) + (\sin \theta)\mathbf{N} \times (\mathbf{E} - \mathbf{S})$$

The weighted distance as a function of the single joint angle is

$$D(\theta) = w_H |\mathbf{h}(\theta) - \mathbf{G}_H|^2 + w_E |\mathbf{e}(\theta) - \mathbf{G}_E|^2$$

The minimum occurs when the derivative is zero,

$$D'(\theta) = w_H (\mathbf{h}(\theta) - \mathbf{G}_H) \cdot \mathbf{h}'(\theta) + w_E (\mathbf{e}(\theta) - \mathbf{G}_E) \cdot \mathbf{e}'(\theta) = 0$$

If $x_0 = \cos \theta$ and $x_1 = \sin \theta$, the equation $D'(\theta) = 0$ is clearly a quadratic polynomial in x_0 and x_1 . Moreover, $x_0^2 + x_1^2 = 1$, another quadratic polynomial. The common solutions can be obtained by computing the resultant polynomial (see Section 16.1.2) in x_0 , a quartic polynomial. This can be solved by using closed-form equations or by using iterative polynomial root finders. Similar algorithms can be developed for line or plane goals and for sliding joints.

Tree Manipulator

The situation can be even more complicated. The manipulator can be a tree of line segments. The leaf nodes are typically end effectors, with each leaf having a primary goal. Interior nodes can also be tagged as end effectors with secondary goals. The method of solution is similar to that of list manipulators with multiple end effectors.

Other Variations

Manipulator joints can have their parameters restricted (limited rotation or sliding). The algorithms mentioned earlier must be modified to support this. A joint can be set up to be springlike so that it tends to move toward a specified resting point. A joint can be damped to resist motion, either in a constant fashion or in a limiting fashion where the damping increases with the number of iterations of the joint optimizers.

The implementation must also decide on the order of processing joints. A general implementation will allow the application to select the order. For a list, the two basic orderings are initial joint to final joint or final joint to initial joint. For a tree, the basic orderings are depth-first traversal, breadth-first traversal, or iteration over the leaves and a traversal from each leaf to the root.

Finally, it is possible to specify that a joint cannot change. The initial point of the manipulator always has this property. If an interior joint of a list is tacked down, then the two sublists are in effect separate manipulators, but the first one that connects the two tacked-down joints does not change. Thus, the interior joint acts as the initial point for a smaller manipulator.

5.4 SKINNING

Skin-and-bones animation, or simply *skinning*, is the process of attaching a deformable mesh to a skeletal structure in order to smoothly deform the mesh as the bones move. The skeleton is represented by a hierarchy of *bones*, each bone positioned in the world by a translation and orientation. The *skin* is represented by a triangle mesh for which each vertex is assigned to one or more bones and is a weighted average of the world positions of the bones that influence it. As the bones move, the vertex positions are updated to provide a smooth animation. Figure 5.2 shows a simple configuration of two bones and five vertices.

The intuition of Figure 5.2 should be clear: Each vertex is constructed based on information relative to the bones that affect it. To be more precise, associate with bone B_i the uniform scale s_i , the translation vector $\mathbf{T}_i$, and the rotation matrix R_i . Let the vertex $\mathbf{V}_j$ be influenced by n_j bones whose indices are k_1 through k_{n_j} . The vertex has two quantities associated with bone B_{k_i} : an offset from the bone, denoted Θ_{jk_i} and measured in the model space of the bone, and a weight of influence, denoted w_{jk_i} . The world-space contribution by B_{k_i} to the vertex offset is

$$s_{k_i} R_{k_i} \Theta_{jk_i} + \mathbf{T}_{k_i}$$

This quantity is the transformation of the offset from the bone's model space to world space. The world-space location of the vertex is the weighted sum of all such contributions,

$$\mathbf{V}_j = \sum_{i=1}^{n_j} w_{jk_i} \left(s_{k_i} R_{k_i} \Theta_{jk_i} + \mathbf{T}_{k_i} \right) \quad (5.18)$$

Skinning is supported in current graphics hardware. The `SkinController` in Wild Magic is implemented originally to use the CPU to do all the algebraic calculations. However, there is a sample application on the CD-ROM to show how to do skinning on the GPU. A class can be created to encapsulate the hardware skinning, but it should be based on the `Effect` class because it has a shader program to feed to the renderer. The `Effect` class itself can have a `Controller` object attached to it whose job it is to update the bone matrices.

The relationship between the bones and the vertices can be thought of as a matrix of weights and offsets whose rows correspond to the vertices and whose columns correspond to the bones. For example, given three vertices and four bones, then the array shown in Table 5.3 illustrates a possible relationship. An entry of $\emptyset$ indicates that the bone does not influence the vertex. In this case the implied weight is 0. The indexing in the matrix is the same one referenced in Equation (5.18). The skin vertices are

$$\mathbf{V}_0 = w_{00} (s_0 R_0 \Theta_{00} + \mathbf{T}_0) + w_{02} (s_2 R_2 \Theta_{02} + \mathbf{T}_2) + w_{03} (s_3 R_3 \Theta_{03} + \mathbf{T}_3)$$

$$\mathbf{V}_1 = w_{11} (s_1 R_1 \Theta_{11} + \mathbf{T}_1) + w_{13} (s_3 R_3 \Theta_{13} + \mathbf{T}_3)$$

$$\mathbf{V}_2 = w_{20} (s_2 R_2 \Theta_{20} + \mathbf{T}_2) + w_{23} (s_3 R_3 \Theta_{23} + \mathbf{T}_3)$$

The assumption is that the world transformations of the bones are current. The bones affecting a skin are normally stored in a subtree of the scene hierarchy. This is particularly true when the skin is for a biped model, and the hierarchy represents the biped anatomy. In the depth-first traversal from an `UpdateGS` call, the bones must be visited first before their associated skin is visited.

The skin vertices are constructed in the world coordinates of the bone tree. Any nonidentity local or world transformations stored at the node to which the skin is attached will cause it to be transformed out of that coordinate system—an error. The `Spatial` class of Wild Magic provides the ability to prevent such updates of a

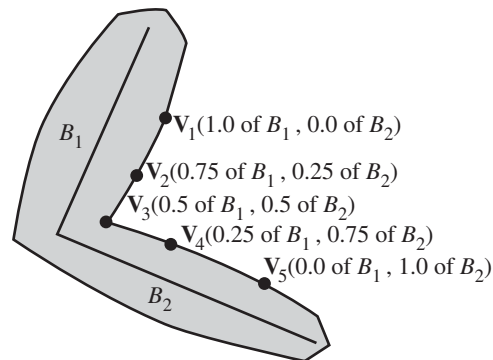


Figure 5.2 A skin-and-bones system consisting of two bones that influence five vertices. The vertex closest to the joint formed by the two bones is equally influenced by the bones. For each vertex farther from the joint, one bone influences it more than the other bone.

Table 5.3 Sample matrix of weights and offsets for skinning.

	B_0	B_1	B_2	B_3
$\mathbf{V}_0$	w_{00}, Θ_{00}	$\emptyset$	w_{02}, Θ_{02}	w_{03}, Θ_{03}
$\mathbf{V}_1$	$\emptyset$	w_{11}, Θ_{11}	$\emptyset$	w_{13}, Θ_{13}
$\mathbf{V}_2$	w_{20}, Θ_{20}	$\emptyset$	$\emptyset$	w_{23}, Θ_{23}

controller by querying the controller if it wants (or does not want) its transformations updated outside the controller.

One final word. Modeling packages have the concept of the *binding pose* for a skinned biped character. The bone transformations, weights, and skin vertex information must be initialized from the binding pose. This step is done in the exporters that convert the modeling package data to Wild Magic format; it is not done in the skin controller class.

5.5 VERTEX MORPHING

The definitions of *morphing* are many. Most of them have the flavor of modifying one object in some well-defined manner to look like another object. The version of morphing that I have implemented as a controller involves a sequence of geometric objects, called *targets*, all having the same number of vertices and mesh topology. Given a vertex on one object, there are corresponding vertices on all the other objects. A weighted average of each collection of corresponding vertices is computed; the result is an object that is a weighted combination of the targets. Think of the weights as an array of numbers summing to 1. The array of weights is applied to *all* sets of corresponding vertices. A set of weight arrays is provided for the morphing, and each set is assigned a time. These act as keyframes: An artist has provided the weights to be used on the objects at a small number of snapshots in time, and a morphing controller does the in-betweening by interpolating the weight arrays and applying the resulting weight array to the set of targets to produce the in-between object.

The Wild Magic class `MorphController` implements this concept. It is provided with the number of vertices in a target, the number of targets, and the number of keys for the weight arrays. Suppose there are V vertices, T targets, and K keys. A weight array has elements w_i for $0 \leq i \leq T-1$ with $w_i \geq 0$ and $\sum_{i=0}^{T-1} w_i = 1$. If $\mathbf{X}_i$ are the set of corresponding vertices to be weighted, with $\mathbf{X}_i$ from target i , then the output vertex is

$$\mathbf{X} = \sum_{i=0}^{T-1} w_i \mathbf{X}_i$$

Observing that $w_{T-1} = 1 - \sum_{i=0}^{T-2} w_i$, the expression is rewritten as

$$\mathbf{X} = \mathbf{X}_0 + \sum_{i=1}^{T-1} w_i (\mathbf{X}_i - \mathbf{X}_0)$$

If the differences $\mathbf{X}_i - \mathbf{X}_0$ are precomputed, then the new expression requires three less multiplications than the old one. The storage requirements are also slightly less: one floating-point value per array of weights since we do not need to store w_0 . For a large amount of morphing data and a lot of keys, this small difference can add up to a large one, both in memory and speed.

An artist can generate all T targets, but an exporter from the modeling package or an importer can be written to precompute the vector differences. The `MorphController` class does assume the precomputing has happened. It is given an array of T vertex arrays; each vertex array is the geometric data of the target and has V vertices. The zeroth vertex array stores the original target. The vertices are the $\mathbf{X}_0$ in the weighted-average equation. The remaining vertex arrays store the vector differences $\mathbf{X}_i - \mathbf{X}_0$.

The morph controller also is given an array of K weight arrays. Each array represents weights w_1 through w_{T-1} , so each array stores $T - 1$ floating-point values. The weights w_0 are not stored. The keyframe times are stored in an array of K floating-point values.

The controller update function takes the input application time and must look up the pair of consecutive keys that bound that time. The process is identical to the one used in keyframe controllers. Time coherency allows an $O(1)$ lookup by saving the index of the first key of the bounding pair found in the last update call and then using it as the starting point for a linear search in the current update call. If t_0 and t_1 are the keyframe times and t is the control time, the normalized time used for the interpolation is $u = (t - t_0)/(t_1 - t_0)$.

5.6 PARTICLE SYSTEMS

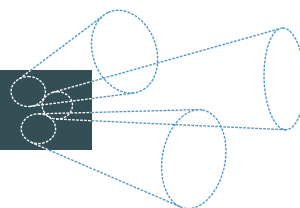
Simple interfaces are provided for controlling a collection of points (`PointController`) or particles (`ParticleController`). The set of points is referred to as a *system*. That system, when viewed as a single entity, moves according to its linear velocity and rotates according to its angular velocity. In physics simulations where the points represent a rigid body, the origin of the system is chosen to be the center of mass of the points, and the coordinate axes are chosen to be the principal directions of the inertia tensor. The controller interface lets you set the linear velocity as a linear speed, `SystemLinearSpeed`, and the unit-length direction, `SystemLinearAxis`. The angular velocity is set by choosing the angular speed, `SystemAngularSpeed`, and the unit-length rotation axis, `SystemAngularAxis`.

In a nonrigid system, each point can have its own linear and angular velocity. These are set by the member functions that expose the arrays of quantities, `PointLinearSpeed`, `PointLinearAxis`, `PointAngularSpeed`, and `PointAngularAxis`. The arrays have the same number of elements as the number of points or particles that are managed by the controller.

The important functions to override in a derived class are `UpdateSystemMotion` and `UpdatePointMotion`. The `Update` function of `PointController` computes the control time from the application time and then calls the two motion updates with the control time. `PointController` provides implementations, but they may be overridden. The system motion update changes the local translation by computing how far the system has moved in the direction of linear velocity and adding it to the current

local translation. Similarly, the local rotation is updated by multiplying it by the incremental rotation implied by the angular speed and angular axis. The local translation of each point is updated using the distance traveled by the point in the direction of its linear velocity. The point does not have a size, so how do you interpret angular velocity? The point could be a summary statistic of an object that does have size. Each point may have a normal vector assigned to it. If normal vectors are attached to the points, those vectors are rotated by the new local rotation matrix.

The base-level support for points and particles is minimal. The number of physics effects you can apply to particle systems is as unlimited as the number of graphics effects you can apply to meshes via shader programming. Each effect has its own special set of requirements. Chapter 9 has some examples, and the CD-ROM has quite a few sample applications to illustrate. A detailed discussion of physics itself and the application to real-time games is a large topic. For example, see [Ebe03].



SPATIAL SORTING

The classic reason for geometric sorting is for correct drawing of objects, both opaque and semitransparent. The opaque objects should be sorted from front to back, based on an observer's location, and the semitransparent objects should be sorted from back to front. The sorted opaque objects are drawn first, and the sorted semitransparent objects are drawn second.

Geometric sorting is not the only important reason for reorganizing your objects. In many situations, changes in the render state can cause the renderer to slow down. The most obvious case is when you have a limited amount of VRAM and more textures than can fit in it. Suppose you have a sequence of six objects to draw, S_1 through S_6 , and each object has one of three texture images assigned to it. Let I_1 , I_2 , and I_3 be those images; assume they are of the same size and that VRAM is limited in that it can only store two of these at a time. Suppose the order of objects in the scene leads to the images being presented to the renderer in the order $I_1, I_2, I_3, I_1, I_2, I_3$. To draw S_1 , I_1 is loaded to VRAM and the object is drawn. Image I_2 is then loaded to VRAM and S_2 is drawn. To draw S_3 , image I_3 must be loaded to VRAM. There is no room for it, so one of the images must be discarded from VRAM. Assuming a least frequently used algorithm, I_1 is discarded. At that point I_3 is loaded, in which case VRAM stores I_2 and I_3 , and S_3 is drawn. S_4 requires I_1 to be loaded to VRAM. That image was just discarded, so it needs to be loaded again. Since I_2 is the least frequently used, it is discarded and I_1 is loaded. Now VRAM stores I_3 and I_1 . S_4 may be drawn. S_5 requires I_2 to be in VRAM. Once again we have the undesirable situation of having to reload an image that was just discarded. When all six objects have been drawn, VRAM has performed four discard operations. Since sending large textures across the memory bus to the graphics card is expensive, the discards can really reduce the frame rate.

If we were instead to sort the objects by the images that they use, we would have S_1 , S_4 , S_2 , S_5 , S_3 , and S_6 . Image I_1 is loaded to VRAM, and S_1 is drawn. We can immediately draw S_4 since it also uses I_1 and that image is already in VRAM. Image I_2 is loaded to VRAM, and both S_2 and S_5 are drawn. In order to handle the last two objects, VRAM must discard I_1 , load I_3 , and then draw S_3 and S_6 . In this drawing pass, only one discard has occurred. Clearly, the sorting by texture image buys you something in this example.

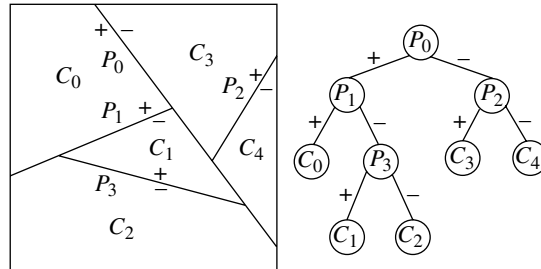
In general, if your profiling indicates that a frequent change in a specific render state is a bottleneck, sorting the objects by that render state should be beneficial. You set the render state once and draw all the objects.

The five sections in this chapter are about geometric sorting. The first topic is sorting of spatial regions using binary space partitioning trees (BSP trees). The BSP trees are not used for partitioning triangle meshes. The second topic is sorting the children at a node. Since a drawing pass uses a depth-first traversal, the order of the children is important. The third topic is portals, an automatic method to cull nonvisible geometric objects. The fourth topic is user-defined maps. Essentially, the game designers and developers partition the world, saving the information for use by the game application. The fifth topic is a simple mechanism to support culling of occluded objects.

6.1 BINARY SPACE PARTITIONING TREES

A popular sorting method is *binary space partitioning*, in which n -dimensional space is recursively partitioned into convex subsets by hyperplanes. For $n = 2$, the partitioning structure is a line; for $n = 3$, the partitioning structure is a plane. A *binary space partitioning tree*, or *BSP tree*, is the data structure used to represent the partitioning. For $n = 3$, the root node represents all of space and contains the partitioning plane that divides space into two subsets. The first child (front child, left child) represents the subset corresponding to that portion of space on the positive side of the plane. That is, if the partitioning plane is $\mathbf{N} \cdot \mathbf{X} - d = 0$, then the left child represents those points for which $\mathbf{N} \cdot \mathbf{X} - d > 0$. If the partitioning plane is generated by a face of an object, and if the eye point is on the positive side of the plane, then the face is visible and is called front facing.

The second child (back child, right child) represents the subset corresponding to the negative side of the plane. Either of the subsets can be further subdivided by other planes, in which case those nodes store the partitioning plane and their children represent yet smaller convex subsets of space. The leaf nodes represent the final convex sets in the partition. These sets can be bounded or unbounded. Figure 6.1 illustrates a BSP tree in two dimensions. The square is intended to represent all of the xy -plane. The interior nodes of the tree indicate which planes they represent, and the leaf nodes indicate which convex regions of space they represent. The first formal papers on this topic were [FKN79, FKN80]. The BSP FAQ [Wad] provides a

Figure 6.1 BSP tree of the xy -plane.

good summary of the topic and has links to websites containing other information or source code.

6.1.1 BSP TREE CONSTRUCTION

Although a BSP tree is a partitioning of space, it may also be used to partition objects in space. If an object is on the positive side of a partition plane, then that object is associated with the front child of the node representing the plane. Similarly, if an object is on the negative side of the plane, it is associated with the back child. The difficulty in classification occurs when the object straddles the plane. In this case the object can be split into two sub-objects, each associated with a child node. If the objects are polytopes, then the sub-objects are also polytopes that share a common face on the partition plane. An implementation of BSP trees that treats the objects in the world as a polygon soup may store the common face with the node of the partition plane. Because of the potential to do a lot of splitting, this saves memory since the common face data is stored once and shared by the polytopes. The pseudocode for construction follows. A precondition is that the initial polygon list is not empty.

```
void ConstructTree (BspTree tree, PolygonList list)
{
    PolygonList posList, negList;
    EdgeList sharedList;

    tree.plane = SelectPartitionPlane(list); // Dot(N,X)-c = 0
    for (each polygon in list) do
    {
        type = Classify(polygon,tree.plane);
        if (type == POSITIVE) then
```

```

    {
        // Dot(N,X)-c >= 0 for all vertices with at least
        // one positive.
        posList.Add(polygon);
    }
    else if (type == NEGATIVE) then
    {
        // Dot(N,X)-c <= 0 for all vertices with at least
        // one negative.
        negList.Add(polygon);
    }
    else if (type == TRANSVERSE) then
    {
        // Dot(N,X)-c is positive for at least one vertex
        // and negative for at least one vertex.
        Polygon posPoly, negPoly;
        Edge sharedEdge;
        Split(polygon,tree.plane,posPoly,negPoly,
            sharedEdge);
        posList.Add(posPoly);
        negList.Add(negPoly);
        sharedList.Add(sharedEdge);
    }
    else // type == COINCIDENT
    {
        // Dot(N,X)-c = 0 for all vertices.
        tree.coincident.Add(polygon);
    }
}

if (sharedList is not empty)
{
    // Find all disjoint polygons in the intersection of
    // partition plane with polygon list.
    PolygonList component;
    ComputeConnectedComponents(sharedList,component);
    tree.coincident.Append(component);
}

if (posList is not empty)
{
    tree.positive = new BspTree;
    ConstructTree(tree.positive,posList);
}

```



```

if (negList is not empty)
{
    tree.negative = new BspTree;
    ConstructTree(tree.negative, negList);
}
}

```

The function `SelectPartitionPlane` chooses a partition plane based on what the application wants. The input is the polygon list because typically a plane containing one of the polygons is used, but it is possible to select other planes based on the list data.

The function `Split` for triangle lists uses the clipping algorithm mentioned in Section 3.1.3. More generally, the loop over the polygon list represents the general Boolean operation of splitting a polygonal object by a plane. This allows a BSP tree to be used for computational solid geometry operations. The pseudocode is structured to indicate that the positive and negative polygons in a split share vertices. The shared edges are processed later to compute the polygons of intersection in the partition plane. For many applications, having access to these polygons is not necessary, so the shared edge code can be safely removed.

Finally, note that the recursive call of `ConstructTree` terminates when the corresponding tree node contains only coincident polygons. Other criteria for stopping can be used, such as termination (1) when the number of polygons in a positive or negative list is smaller than an application-specified threshold or (2) when the tree reaches a maximum depth.

6.1.2 BSP TREE USAGE

I mainly use BSP trees to partition the world as a coarse-level sorting, not to partition the data in the world. The basic premise is illustrated in Figure 6.2. A line partitions

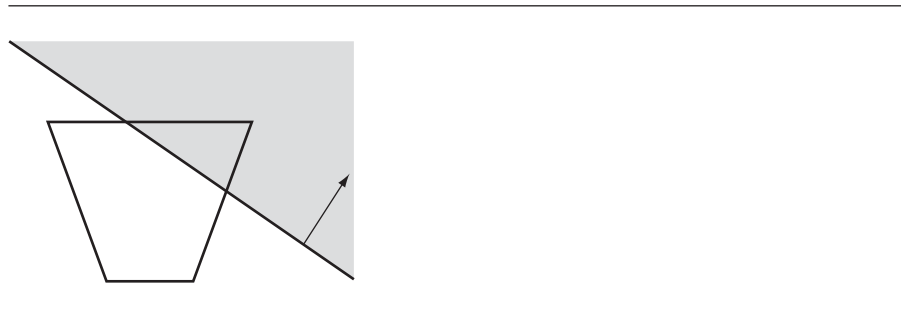


Figure 6.2 An illustration of BSP tree sorting in two dimensions.

the plane into two half planes. The half plane to the side that the line normal points to is gray. The other half plane is white. The view frustum overlaps both half planes. The eye point is in the white half plane. The region that the view frustum encloses is the only relevant region for drawing purposes. If you draw a ray from the eye point to any point inside the gray subregion of the frustum (a line of sight, so to speak), that ray will intersect any objects in the white subregion before it intersects any objects in the gray subregion. Consequently, no object in the gray subregion can occlude an object in the white subregion. If the objects in the gray subregion are drawn so that the depth buffer is correctly written with depth information, you can draw the objects in the white subregion with depth buffering set to write-only. It is not necessary to read the depth buffer for comparisons, because you already know the objects in the white subregion occlude anything in the gray subregion.

A frequent use of BSP trees is where the separating planes are actual geometry in a level, most notably walls, floors, and ceilings. If a wall plane splits space into two half spaces, and if one half space is behind the wall and never visible to the observer, then you need not even draw the region behind the wall. In *Wild Magic*, disabling drawing a half space is accomplished by setting the `Spatial::Culling` flag to `Spatial::CULL_ALWAYS`.

The classical BSP node in a scene graph has a separating plane and two children. One child corresponds to the half space on one side of the plane; the other child corresponds to the other half space. The child subtrees represent those portions of the scene in their respective half spaces. My BSP node stores three children: two to represent the portions of the scene in the half spaces, and the third to represent any geometry associated with the separating plane. For example, in a level with walls, the wall geometry will be part of the scene represented by the third child. The class is `BspNode` and its interface is

```
class BspNode : public Node
{
public:
    BspNode ();
    BspNode (const Plane3f& rkModelPlane);
    virtual ~BspNode ();

    SpatialPtr AttachPositiveChild (Spatial* pkChild);
    SpatialPtr AttachCoplanarChild (Spatial* pkChild);
    SpatialPtr AttachNegativeChild (Spatial* pkChild);
    SpatialPtr DetachPositiveChild ();
    SpatialPtr DetachCoplanarChild ();
    SpatialPtr DetachNegativeChild ();
    SpatialPtr GetPositiveChild ();
    SpatialPtr GetCoplanarChild ();
    SpatialPtr GetNegativeChild ();
```

```

Plane3f& ModelPlane ();
const Plane3f& GetModelPlane () const;
const Plane3f& GetWorldPlane () const;

Spatial* GetContainingNode (const Vector3f& rkPoint);

protected:
    virtual void UpdateWorldData (double dAppTime);
    virtual void GetVisibleSet (Culler& rkCuller, bool bNoCull);

    Plane3f m_kModelPlane;
    Plane3f m_kWorldPlane;
};

```

The class is derived from `Node`. The `BspNode` constructors must create the base class objects. They do so by requesting three children, but set the growth factor to zero; that is, the number of children is fixed at three. The child at index 0 is associated with the positive side of the separating plane; that is, the half space to which the plane normal points. The child at index 2 is associated with the negative side of the separating plane. The child at index 1 is where additional geometry may be attached, such as the triangles that are coplanar with the separating plane. Rather than require you to remember the indexing scheme, the `Attach*`, `Detach*`, and `Get*` member functions are used to manipulate the children.

The separating plane is specified in model-space coordinates for the node. The model-to-world transformations are used to transform that plane into one in world coordinates. This is done automatically by the geometric-state update system via a call to `UpdateGS`, through the virtual function `UpdateWorldData`:

```

void BspNode::UpdateWorldData (double dAppTime)
{
    Node::UpdateWorldData(dAppTime);
    m_kWorldPlane = World.ApplyForward(m_kModelPlane);
}

```

The base class `UpdateWorldData` is called first in order to guarantee that the model-to-world transformation for `BspNode` is up to date.

The `Transformation` class has a member function for transforming a plane in model space to one in world space, namely, `ApplyForward`. Let $\mathbf{X}$ be a point in model space and $\mathbf{Y} = R\mathbf{S}\mathbf{X} + \mathbf{T}$ be the corresponding point in world space, where $\mathbf{S}$ is the diagonal matrix of world scales, R is the world rotation, and $\mathbf{T}$ is the world translation. Let the model-space plane be $\mathbf{N}_0 \cdot \mathbf{X} = c_0$, where $\mathbf{N}_0$ is a unit-length normal vector. The inverse transformation is $\mathbf{X} = S^{-1}R^T(\mathbf{Y} - \mathbf{T})$. Replacing this in

the plane equation and applying some algebra leads to a world plane $\mathbf{N}_1 \cdot \mathbf{Y} = c_1$, where

$$\mathbf{N}_1 = R S^{-1} \mathbf{N}_0 \quad \text{and} \quad c_1 = c_0 + \mathbf{N}_1 \cdot \mathbf{T}$$

If the scale matrix S is not the identity matrix, then $\mathbf{N}_1$ is not unit length. In this case it must be normalized and the constant adjusted:

$$\mathbf{N}'_1 = \frac{\mathbf{N}_1}{|\mathbf{N}_1|} \quad \text{and} \quad c'_1 = \frac{c_1}{|\mathbf{N}_1|}$$

resulting in the world plane $\mathbf{N}'_1 \cdot \mathbf{Y} = c'_1$.

The virtual function `GetVisibleSet` is implemented in `BspNode`. It is designed to compute potentially visible objects according to the description I provided previously in association with Figure 6.2. The source code is

```
void BspNode::GetVisibleSet (Culler& rkCuller, bool bNoCull)
{
    if (m_kEffects.size() > 0)
    {
        // This is a global effect. Place a 'begin' marker
        // in the visible set to indicate the effect is active.
        rkCuller.GetVisibleSet().Insert(this, m_kEffects[0]);
    }

    // Get visible Geometry in back-to-front order. If a
    // global effect is active, the Geometry objects in the
    // subtree will be drawn using it.
    SpatialPtr spkPChild = GetPositiveChild();
    SpatialPtr spkCChild = GetCoplanarChild();
    SpatialPtr spkNChild = GetNegativeChild();

    const Camera* pkCamera = rkCuller.GetCamera();
    int iLocSide = m_kWorldPlane.WhichSide(
        pkCamera->GetLocation());
    int iFruSide = rkCuller.WhichSide(m_kWorldPlane);

    if (iLocSide > 0)
    {
        // Camera origin on positive side of plane.

        if (iFruSide <= 0)
        {
            // The frustum is on the negative side of the
            // plane or straddles the plane. In either case,
```

```

        // the negative child is potentially visible.
        if (spkNChild)
        {
            spkNChild->OnGetVisibleSet(rkCuller,bNoCull);
        }
    }

    if (iFruSide == 0)
    {
        // The frustum straddles the plane. The coplanar
        // child is potentially visible.
        if (spkCChild)
        {
            spkCChild->OnGetVisibleSet(rkCuller,bNoCull);
        }
    }

    if (iFruSide >= 0)
    {
        // The frustum is on the positive side of the plane
        // or straddles the plane. In either case, the
        // positive child is potentially visible.
        if (spkPChild)
        {
            spkPChild->OnGetVisibleSet(rkCuller,bNoCull);
        }
    }
}
else if (iLocSide < 0)
{
    // Camera origin on negative side of plane.

    if (iFruSide >= 0)
    {
        // The frustum is on the positive side of the plane
        // or straddles the plane. In either case, the
        // positive child is potentially visible.
        if (spkPChild)
        {
            spkPChild->OnGetVisibleSet(rkCuller,bNoCull);
        }
    }
}

if (iFruSide == 0)

```

```

    {
        // The frustum straddles the plane. The coplanar
        // child is potentially visible.
        if (spkCChild)
        {
            spkCChild->OnGetVisibleSet(rkCuller,bNoCull);
        }
    }

    if (iFruSide <= 0)
    {
        // The frustum is on the negative side of the plane
        // or straddles the plane. In either case, the
        // negative child is potentially visible.
        if (spkNChild)
        {
            spkNChild->OnGetVisibleSet(rkCuller,bNoCull);
        }
    }
}
else
{
    // Camera origin on plane itself. Both sides of the
    // plane are potentially visible as well as the plane
    // itself. Select the first-to-be-drawn half space to
    // be the one to which the camera direction points.
    float fNdD = m_kWorldPlane.Normal.Dot(
        pkCamera->GetDVector());
    if (fNdD >= 0.0f)
    {
        if (spkPChild)
        {
            spkPChild->OnGetVisibleSet(rkCuller,bNoCull);
        }

        if (spkCChild)
        {
            spkCChild->OnGetVisibleSet(rkCuller,bNoCull);
        }

        if (spkNChild)
        {
            spkNChild->OnGetVisibleSet(rkCuller,bNoCull);
        }
    }
}

```

```

else
{
    if (spkNChild)
    {
        spkNChild->OnGetVisibleSet(rkCuller,bNoCull);
    }

    if (spkCChild)
    {
        spkCChild->OnGetVisibleSet(rkCuller,bNoCull);
    }

    if (spkPChild)
    {
        spkPChild->OnGetVisibleSet(rkCuller,bNoCull);
    }
}

if (m_kEffects.size() > 0)
{
    // Place an 'end' marker in the visible set to
    // indicate that the global effect is inactive.
    rkCuller.GetVisibleSet().Insert(0,0);
}
}

```

The three children must be drawn in back-to-front order. It is possible that any of the three children have empty subtrees, so the smart pointers for those children must be tested to see if they are not null before using them.

The first step, of course, is to determine on which side of the separating plane the eye point is located. This is the role of the code

```

const Camera* pkCamera = rkCuller.GetCamera();
int iLocSide = m_kWorldPlane.WhichSide(
    pkCamera->GetLocation());
int iFruSide = rkCuller.WhichSide(m_kWorldPlane);

```

As Figure 6.2 indicates, we also need to know how the view frustum is positioned relative to the separating plane. The Plane class has a member function `WhichSide` that determines whether the input point is on the positive side of the plane (return value is positive), on the negative side of the plane (return value is negative), or on the plane (return value is zero). The Culler class has a member function `WhichSide` that tests the eight vertices of the view frustum to see on which side of the plane they lie. If all eight lie on the positive side of the plane, the return value is positive. If all eight

lie on the negative side of the plane, the return value is negative. Otherwise, some of the eight lie on the positive side and some lie on the negative side, and the function returns zero.

Consider the block of code for when the eye point is on the negative side of the plane. This is the configuration in Figure 6.2. If the view frustum is on the positive side of the plane or straddles the plane, the gray subregion must be drawn first. This is the positive child of the BSP node. As you can see in the code, that child is drawn first. If the frustum is fully on the positive side, then the separating plane does not cut through it, so any geometry associated with that plane need not be drawn. If the separating plane does intersect the frustum, then you should draw the geometry for the plane (if any). The code block that compares `iFruSide` to zero handles this. Naturally, when the frustum straddles the plane, you also need to draw the negative child. That is the last code block in the clause that handles the eye point on the negative side of the plane.

What to do when the eye point is exactly on the separating plane appears to be a technical complication. For an environment where you have walls as the separating planes, you would actually prevent this case, either by some prior knowledge about the structure of the environment and the eye point location or by a collision detection system. As it turns out, there is nothing to worry about here. Any ray emanating from the eye point through the frustum is either fully on one side of the plane, fully on the other side, or in the plane itself. In my code, though, I choose to order the drawing of the children based on the half space that contains the camera view direction.

In the code block for when the eye point is on the negative side of the plane, the view frustum straddles the plane, and the BSP node has three children, all the children will be drawn. In the example of an environment where the plane of a wall is used as the separating plane, the child corresponding to the nonvisible half space does not need to be drawn. You, the application writer, must arrange to set the `Culling` flag to `CULL_ALWAYS` for that child so that the drawing pass is not propagated down the corresponding subtree. That said, it is possible that the camera moves along the wall to a doorway that does let you see into the space behind the wall. In this case you need to structure your application logic to set the `Culling` flag according to the current location of the eye point. This is the stuff of occlusion culling in a game, essentially keeping a map of the world that helps you identify which objects are, or are not, visible from a given region in the world.

The leaf nodes of a BSP tree implicitly represent a region of space that is convex. The region is potentially unbounded. Many times it is useful to know which of these convex regions a point is in. The function

```
Spatial* GetContainingNode (const Vector3f& rkPoint);
```

is the query that locates that region. The return value is not necessarily of type `BspNode`. The leaf nodes of the BSP tree can be any `Spatial`-derived type you prefer.

6.2 NODE-BASED SORTING

One of the simplest, coarse-level sorting methods to be applied in a scene hierarchy is to sort the children of a node. How they are sorted depends on what your environment is and how the camera is positioned and oriented relative to the children of the node.

To demonstrate, consider the example of a node that has six `TriMesh` objects that are the faces of a cube. The faces are textured and semitransparent, so you can see the back faces of the cube through the front faces. The global state is set as indicated:

- Back-face culling is disabled. Because each face is semitransparent, you must be able to see it when positioned on either side of the face.
- Depth buffering is enabled for writing, but not reading. The faces will be depth sorted based on the location of the eye point, and then drawn in the correct order. Reading the depth buffer to determine if a pixel should be drawn is not necessary. For the cube only, it is not necessary to write to the depth buffer. If other objects are drawn in the same scene using depth buffering with reading enabled, you need to make certain that the depths are correct. That is why writing is enabled.
- Alpha blending is enabled at the node since the face textures have alpha values to obtain the semitransparency.

The cube is constructed in its model space to have its center at the origin $(0, 0, 0)$. The faces perpendicular to the x -axis are positioned at $x = 1$ and $x = -1$. The faces perpendicular to the y -axis are positioned at $y = 1$ and $y = -1$. The faces perpendicular to the z -axis are positioned at $z = 1$ and $z = -1$. The camera is inverse transformed from the world into the model space of the cube. The back faces and front faces are determined solely by analyzing the components of the camera view direction in the cube's model space. Let that direction be $\mathbf{D} = (d_0, d_1, d_2)$. Suppose that the eye point is at $(2, 0, 0)$ and you are looking directly at the face at $x = 1$. The view direction is $(-1, 0, 0)$. The $x = 1$ face is front facing. Its outer normal is $(1, 0, 0)$. The angle between the view direction and the outer normal is greater than 90 degrees, so the cosine of the angle is negative. The dot product of the view direction and outer normal is the cosine of the angle. In the current example, the dot product is $(-1, 0, 0) \cdot (1, 0, 0) = -1 < 0$. The $x = -1$ face is back facing. Its outer normal is $(-1, 0, 0)$. The cosine of the angle between the view direction and the outer normal is $(-1, 0, 0) \cdot (-1, 0, 0) = 1 > 0$. Similar arguments apply to the other faces. The classification for back faces is summarized by

- $d_0 > 0$: Face $x = 1$ is back facing.
- $d_0 < 0$: Face $x = -1$ is back facing.
- $d_1 > 0$: Face $y = 1$ is back facing.

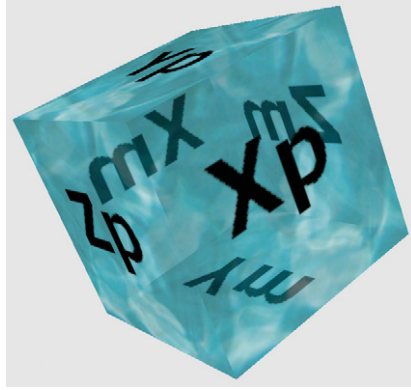


Figure 6.3 A cube with transparent faces. The faces are sorted so that the drawing order produces the correct rendering.

- $d_1 < 0$: Face $y = -1$ is back facing.
- $d_2 > 0$: Face $z = 1$ is back facing.
- $d_2 < 0$: Face $z = -1$ is back facing.

A sorting algorithm for the faces will inverse-transform the camera's world view direction to the model space of the cube; the resulting direction is (d_0, d_1, d_2) . The signs of the d_i are tested to determine the cube faces that are back facing. The six children of the node are reordered so that the back faces occur first and the front faces occur second. A sample application, `SampleGraphics/SortFaces`, implements the algorithm described here. A screen shot is shown in Figure 6.3.

6.3 PORTALS

The portal system is designed for indoor environments where you have lots of regions separated by opaque geometry. The system is a form of occlusion culling and attempts to draw only what is visible to the observer. The regions form an abstract graph. Each region is a node of the graph. Two regions are adjacent in the graph if they are adjacent geometrically. A *portal* is a doorway that allows you to look from one region into another region adjacent to it. The portals are the arcs for the abstract graph. From a visibility perspective, a portal is bidirectional. If you are in one region and can see through a doorway into an adjacent room, then an observer in the adjacent region should be able to look through the same doorway into the original region. However, you can obtain more interesting effects in your environment by making portals unidirectional. The idea is one of teleportation. Imagine a region that exists

in one “universe” and allows you to look through a portal into another “universe.” Once you step through the portal, you turn around and look back. The portal is not there! I am certain you have seen this effect in at least one science-fiction movie. The Wild Magic engine implements portals to be unidirectional.

The portal system is also a form of sorting in the following sense. The drawing pass starts in one region. The standard depth-first traversal of the subscene rooted at the region node is bypassed. Instead, the drawing call is propagated to regions that are adjacent to the current one and that are visible through portals. Effectively, the regions are sorted based on visibility. Suppose you have three regions (*A*, *B*, and *C*) arranged along a linear path, each having portals into the adjacent regions. If you are in region *A* and can see through a portal to *B*, and you can additionally see through a portal in *B* to the region *C*, then *C* is the farthest region you can see from your current position. Region *C* should be drawn first, followed by region *B*, and then your own region *A*. The drawing pass must be careful to prevent cycles in the graph. The system does have Boolean flags to tag regions whenever they have been visited. These flags prevent multiple attempts to draw the regions.

The Wild Magic portal system uses a BSP tree to decompose the indoor environment. The leaf nodes of the BSP tree are convex regions in space. The class `ConvexRegion` is derived from `Node` and is used to represent the leaf nodes. Any geometric representation for the region, including walls, floors, ceilings, furniture, or whatever, may be added as children of the convex region node. The root of the BSP tree is a special node that helps determine in which leaf region the eye point is. Another class is designed to support this, namely, `ConvexRegionManager`. It is derived from `BspNode`. The middle child of such a node is used to store the representation for the outside of the encapsulated region, just in case you should choose to let the player exit your indoor environment. Finally, the class `Portal` encapsulates the geometry of the portal and its behavior. The abstract graph of regions is a collection of `ConvexRegion` objects and `Portal` objects. Both types of objects have connections that support the graph arcs.

Figure 6.4 illustrates the basic graph connections between regions and portals. The outgoing portals for the convex region in the figure can, of course, be the incoming portals to another convex region, hence the abstract directed graph. Figure 6.5

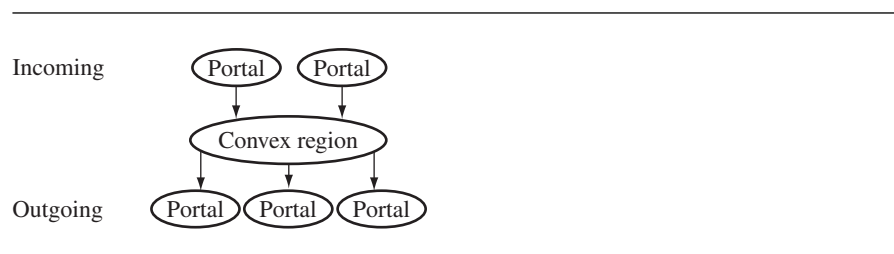


Figure 6.4 A `ConvexRegion` node. The portals with arrows to the node are the *incoming portals* to the region. The arrows from the node to the other portals are *outgoing portals*.

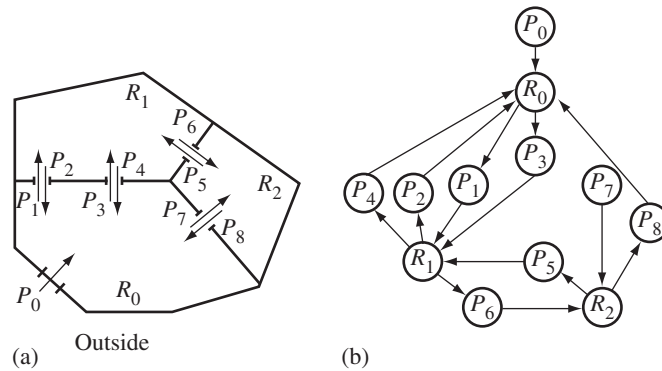


Figure 6.5 A configuration of three regions, nine portals, and an outside area. (a) The geometric layout for the regions and portals. (b) The directed graph associated with it.

shows a specific configuration of regions and portals, including the directed graph associated with it.

The portal P_0 is from the outside to inside the collection of regions. The portal is displayed as if the door were closed. Once a player-character stands in front of the door, a mouse click can open it. The character steps through, and the door closes behind him (never to open again). The other doorways each have two unidirectional portals, so no teleportation occurs in this configuration.

The occlusion culling comes into play as follows. Figure 6.6 shows two regions with a portal from one region to the other. Using the standard drawing with a frustum, the renderer will draw everything in the gray region shown in part (a) of the figure, including the object shown as a small, black dot. That object is not visible to the observer, but the renderer did not know this until too late, that is, when the depth buffer comparisons showed that the wall is closer to the eye point and occludes the object.

In previous versions of Wild Magic, I required that the portal polygons be convex. The planes formed by the eye point and the edges of the polygon formed a convex solid. Objects in the adjacent room that are outside this solid are not visible, hence they are culled. This is an aggressive form of culling where you hope to reduce the drawing time by culling objects in exchange for more time doing test-intersection queries between bounding volumes and planes. The hope is that the net time is shorter than if you did not do portal culling. The problem is that the more edges the portals have, the more planes you have to test against. Given the speed of current graphics hardware, a less aggressive form of culling is better to use. Rather than add a culling plane per edge of the portal, you should always use a fixed number of culling planes. The view frustum itself has six planes: near, far, left, right, bottom, and top.

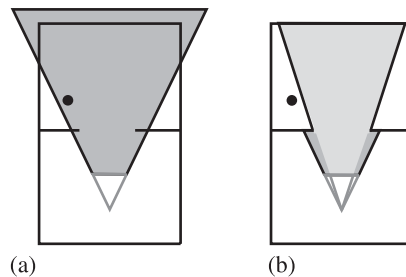


Figure 6.6 Two regions with a portal from one to the other. (a) The gray region depicts the view frustum. (b) The trimmed version of the view frustum using planes formed by the eye point and edges of the portal polygon.

The reduced frustum used to draw the adjacent room will also have six planes: the near and far planes from the original frustum but four planes replacing the left, right, bottom, and top original frustum planes. The idea is to create a bounding rectangle that contains the portal and use its edges to build the other four planes. This allows the original portal polygon to be any simple polygon of arbitrary complexity. And it guarantees a fixed amount of time spent on test-intersection queries between frustum planes and bounding volumes.

The smaller frustum used for the adjacent region is shown as light gray in Figure 6.6. Keep in mind that the smaller frustum is used only for culling. The regular view frustum is still used for drawing, so the renderer may attempt to draw portions of the walls in the adjacent region, even though they are partially occluded. The idea is to eliminate occluded objects from the drawing pass. You could design the camera system to tell the graphics system to use the additional culling planes for clipping, but that has the potential to take away resources from other objects. The current consumer graphics hardware is powerful enough that you might as well just let it go ahead and clip.

At the top level of the system, we have class `ConvexRegionManager`. Its interface is

```
class ConvexRegionManager : public BspNode
{
public:
    ConvexRegionManager ();

    SpatialPtr AttachOutside (Spatial* pkOutside);
    SpatialPtr DetachOutside ();
    SpatialPtr GetOutside ();
};
```

```

        ConvexRegion* GetContainingRegion (const Vector3f& rkPoint);

protected:
    virtual void Draw (Renderer& rkRenderer, bool bNoCull = false);
};

```

A *convex region manager* is a BSP tree whose leaf nodes are the `ConvexRegion` objects. A subscene representing the outside of the environment, if any, can be attached or detached via the member functions `AttachOutside` and `DetachOutside`. The outside scene can be as complex as you like, especially so if you plan on an application that has both an outdoor and an indoor environment.

The main role of `ConvexRegionManager` is to locate the convex region that contains the eye point. The function `GetContainingRegion` supports this query. If the function returns `NULL`, the eye point is not in any of the convex regions and, for all practical purposes, is outside. The `GetVisibleSet` function used for culling is fairly simple:

```

void ConvexRegionManager::GetVisibleSet (Culler& rkCuller, bool bNoCull)
{
    ConvexRegion* pkRegion = GetContainingRegion(
        rkCuller.GetCamera()->GetLocation());

    if (pkRegion)
    {
        // Accumulate visible objects starting in the region
        // containing the camera.
        pkRegion->GetVisibleSet(rkCuller,bNoCull);
    }
    else
    {
        // The camera is outside the set of regions.
        // Accumulate visible objects for the outside scene
        // (if it exists).
        if (GetOutside())
        {
            GetOutside()->GetVisibleSet(rkCuller,bNoCull);
        }
    }
}

```

A situation you must guard against in your application is the one where the eye point is outside, but the near plane of the view frustum straddles a separating wall between inside and outside. The convex region manager determines that the eye point is outside, so the region traversal for culling is never initiated. The outside is drawn,

but not correctly because the view frustum contains part of the inside environment that never gets drawn.

The only reason I have `ConvexRegionManager` in the engine is to provide an automatic method for locating the convex region containing the eye point. The containment query is called in each drawing pass, even if the eye point has not moved. Since the object is a BSP tree, presumably with a small height, the cost of the query should not be an issue. However, if you were to keep track of the eye point and containing room through other means, say, by a map you have of the indoor environment, there is no need for the BSP tree. The graph of `ConvexRegion` and `Portal` objects works just fine without the manager.

The interfaces for the `ConvexRegion` and `Portal` classes are

```
class ConvexRegion : public Node
{
public:
    ConvexRegion (int iPQuantity, Portal** apkPortal);
    virtual ~ConvexRegion ();
    int GetPortalQuantity () const;
    Portal* GetPortal (int i) const;

protected:
    ConvexRegion ();
    virtual void UpdateWorldData (double dAppTime);
    int m_iPQuantity;
    Portal** m_apkPortal;
    bool m_bVisited;

// internal use
public:
    virtual void GetVisibleSet (Culler& rkCuller, bool bNoCull);
};

class Portal : public Object
{
public:
    Portal (int iVQuantity, Vector3f* akModelVertex,
           ConvexRegion* pkAdjacentRegion, bool bOpen);
    virtual ~Portal ();
    ConvexRegion*& AdjacentRegion ();
    bool& Open ();

protected:
    Portal ();
    friend class ConvexRegion;
```

```

void UpdateWorldData (const Transformation& rkWorld);
void GetVisibleSet (Culler& rkCuller, bool bNoCull);

int m_iVQuantity;
Vector3f* m_akModelVertex;
Vector3f* m_akWorldVertex;
Vector3f m_akModelBRect[4];
Vector3f m_akWorldBQuad[4];
ConvexRegion* m_pkAdjacentRegion;
bool m_bOpen;
};

```

The `ConvexRegion` constructor is passed an array of outgoing portals associated with the convex region. The class will use the array pointer directly and will delete the array during destruction. Because the class takes over ownership of the portal array, the portals cannot be shared between convex regions.

The `Portal` constructor is passed an array of vertices that represent the portal geometry, a pointer to the adjacent region (so this portal is incoming for that region), and a Boolean flag indicating whether the portal is initially open or closed. The model-space vertices must be counterclockwise ordered when looking through the portal to the adjacent region, and they must be in the model-space coordinates for the region that contains the portal. A bounding rectangle is computed for the model-space vertices. A world-space bounding quadrilateral is computed from the model-space rectangle during an `UpdateWorldData` call.

`ConvexRegion` overrides the `UpdateWorldData` virtual function in order to update the geometric state in its subtree in the normal manner in which an `UpdateGS` processes the subtree. The outgoing portals themselves might need updating. Since `Portal` is not derived from `Spatial`, these objects are not visited by the `UpdateGS` pass. The convex region must initiate the update of the portals. The source code is

```

void ConvexRegion::UpdateWorldData (double dAppTime)
{
    // Update the region walls and contained objects.
    Node::UpdateWorldData(dAppTime);

    // Update the portal geometry.
    for (int i = 0; i < m_iPQuantity; i++)
    {
        m_apkPortal[i]->UpdateWorldData(World);
    }
}

```

The portal objects must update their own data, and do so with a single batch update:


```

void Portal::UpdateWorldData (const Transformation& rkWorld)
{
    rkWorld.ApplyForward(m_iVQuantity,m_akModelVertex,
        m_akWorldVertex);
}

```

The culling function in both classes is an implementation of the traversal of a directed graph. Because the graph most likely has cycles, the code needs to maintain Boolean flags indicating whether or not a region has already been visited to prevent an infinite loop. The ConvexRegion class has a data member, `m_bVisited`, for this purpose. The drawing routine for a convex region is

```

void ConvexRegion::GetVisibleSet (Culler& rkCuller,
    bool bNoCull)
{
    if (!m_bVisited)
    {
        m_bVisited = true;

        // Add anything visible through open portals.
        for (int i = 0; i < m_iPQuantity; i++)
        {
            m_apkPortal[i]->GetVisibleSet(rkCuller,bNoCull);
        }

        // Add the region walls and contained objects.
        Node::GetVisibleSet(rkCuller,bNoCull);

        m_bVisited = false;
    }
}

```

The convex region manager starts the culling pass in the region containing the eye point. On entry to the culling function for this region, the visitation flag is false. The flag is then set to true to indicate that the region has been visited. The outgoing portals associated with the region are asked to propagate the culling to their adjacent regions. During the propagation, if the current region is revisited, its visitation flag will prevent another recursive call (and avoid the infinite loop). In Figure 6.5, if the eye point is in region R_0 , a cycle is formed by following portal P_1 into R_1 , and then immediately returning to R_0 through portal P_2 . A larger cycle occurs, this one by following P_1 into R_1 , P_6 into R_2 , and then P_8 into R_0 . Once the graph of regions has been traversed, the recursive call comes back to the original region and the `Node::GetVisibleSet` call is made. This call is what tests visibility in the interior of

the region and all its contents. The visitation flag is reset to false to allow the next culling call to the portal system.

You might have noticed that this process has the potential for being very slow. I mentioned that the graph of regions is entirely traversed. In a large indoor environment, there could be a substantial number of regions, most of them not visible. If the portal system visits all the nonvisible regions and attempts to draw them anyway, what is the point? Not to worry. As described earlier, the portals are used to generate additional culling planes for the camera to use. The planes are used to cull objects not visible to the observer, including portals themselves! Return once again to Figure 6.5. Suppose the observer is in region R_0 and standing directly in front of the doorway marked portal P_1 . The observer then looks straight ahead into region R_1 through that portal. The portal planes generated by the observer's eye point and the edges of the portal polygon form a narrow frustum into region R_1 . The portal marked P_6 is not visible to the observer. The portal drawing system will make sure that the region traversal does not continue through P_6 . In this manner, a carefully designed environment will have only a few potentially visible regions along a line of sight, so only a few regions will be processed by the renderer.

The portal culling code is listed next but with pseudocode for the saving (push) and restoring (pop) of culling planes.

```
void Portal::GetVisibleSet (Culler& rkCuller, bool bNoCull)
{
    // Only visit adjacent region if portal is open.
    if (!m_bOpen)
    {
        return;
    }

    // Only traverse through visible portals.
    if (!rkCuller.IsVisible(m_iVQuantity, m_akWorldVertex, true))
    {
        return;
    }

    // Set rkCuller left, right, top, bottom planes from
    // portal's world bounding quadrilateral.
    <code goes here>;

    // Visit the adjacent region and any nonculled objects in it.
    m_pkAdjacentRegion->GetVisibleSet(rkCuller, bNoCull);

    // Restore the left, right, top, bottom planes that
    // were in effect before this GetVisibleSet was called.
    <code goes here>;
}
```

I mentioned that the `Portal` constructor takes a Boolean input that indicates whether or not the portal is “open.” The intent is that if you can see through the portal, it is open. If not, it is closed. In a typical game, a character arrives at a closed door, preventing him from entering a region. A magical click of the mouse button causes the door to pop open, and the character steps into the region. The open flag is used to support this and controls whether or not a portal propagates the `GetVisibleSet` call to the adjacent region. The first step that the `Portal::GetVisibleSet` function takes is to check that Boolean flag.

The second step in the drawing is to check if this portal is visible to the observer. If it is, the reduced frustum is computed using the current eye point and the planes implied by the world bounding quadrilateral for the portal. The adjacent region is then told to build its potentially visible set. Thus, `ConvexRegion::GetVisibleSet` and `Portal::GetVisibleSet` form a recursive chain of functions. Once the region is drawn, the frustum that was active before the current call to `GetVisibleSet` is restored.

If the portal is culled, then the `GetVisibleSet` call is not propagated. In my previous example using Figure 6.5, an observer in region R_0 standing in front of portal P_1 will cause the region traversal to start in R_0 . When portal P_1 has its `GetVisibleSet` function called, the portal is open and the portal itself is visible to the camera, so the reduced frustum is formed and the adjacent region must be processed. A traversal over its outgoing portals is made, and the portals are told to propagate the `GetVisibleSet` call. We will find in `Portal::GetVisibleSet` for P_6 that this portal is not visible to the observer; the planes for P_1 are on the camera’s stack, but the planes for P_6 are not. The `GetVisibleSet` call is not propagated to R_1 (through that path).

6.4 USER-DEFINED MAPS

The user-defined map is the data structure that stores the cells. Each cell is analyzed to determine the set of objects that are potentially visible and the set of objects that are not visible. I have alluded to such a data structure a few times thus far. The idea is to partition the world into cells.

The user-defined map is loaded by the application, and the location of the camera is tracked relative to the map. Every time the camera leaves one cell and enters another, the `Spatial::Culling` flags for the objects in the previous cell are all set to `CULL_ALWAYS`. The objects in the potentially visible set for the new cell have their flags set to `CULL_DYNAMIC` (or `CULL_NEVER` if you prefer to save time and not cull), and the objects that are known not to be visible have their flags set to `CULL_ALWAYS`.

6.5 OCCLUSION CULLING

Depth buffering is effectively an occlusion culling system, but it is performed on a per-pixel basis—an *image-space algorithm*. The portal system is a system that provides

some automatic culling of occluded objects. This is an *object-space algorithm*. The walls of rooms are the *occluders* for an object unless the object is visible (or partially visible) through a portal.

It is possible to attempt culling in a manner that is somewhat the opposite of portals. A portal polygon potentially allows you to see an object that is not occluded by walls. The portion of the wall that remains after you cut out a portal polygon is considered a *blocker*. Generally, a polygon can be a blocker in the sense that if you form a convex polyhedron with a vertex at the eye point and with triangular faces formed by the eye point and the edges of the polygon, any object inside the convex polyhedron is not visible to the observer.

It is possible to design game levels so that certain static geometry can be used as blockers in order to cull objects. For example, imagine an indoor environment that represents an ancient temple. Each room in the temple has thick vertical columns from floor to ceiling. From the observer's perspective, a column looks like a rectangular polygon on the screen. Any object hidden by that rectangle may be culled. It is possible to dynamically construct the culling planes given the camera's current position and the columns' current positions. The `Culler` class may be modified to attempt culling based on this information.

General systems for dynamic occlusion culling are difficult to make fast enough for real-time applications. One system that appeared to be reasonable was *SurRender Umbra*, a product that was released by Hybrid Holdings of Finland in the early 2000s. You can still find references to it online; see, for example, [AM01].

For practical purposes, use of blocker polygons is probably the quickest way to obtain some additional culling. As is often the case, the increased speed and memory capabilities of current graphics hardware outweighs the need for CPU-intensive aggressive culling. Thus, it is usually faster to have a simple culling system and to draw some objects even if they are not visible, rather than spending a lot of CPU time trying to aggressively cull objects.



CHAPTER

LEVEL OF DETAIL

The rendering of a detailed and complex model that consists of thousands of triangles looks quite good when the model is near the eye point. The time it takes to render the large number of triangles is well worth the gain in visual quality. However, when the same model is far from the eye point, the detail provided by thousands of triangles is not that noticeable, because the screen-space coverage of the rendered model might be only a handful of pixels. In this situation, the trade-off in time versus visual quality is not worth it. If the final rendering covers only a handful of pixels, the number of triangles processed should be proportional. This chapter introduces the concept of *geometric level of detail* (LOD). The amount of work done by the renderer per model per pixel should be as independent of the number of triangles that make up the model as possible.

Although rendering time and potential loss of visual quality factor into decisions about level of detail, geometric level of detail can also be important for nonvisual aspects of the game engine, most notably in collision detection. A character in the game might consist of some 10,000 triangles so that the rendered version is visually appealing. If that character is to interact with his environment for purposes of collision, it would be quite expensive to process most (or all) of the 10,000 triangles in an intersection test with a wall of a room. An alternative is to provide one or more coarse resolution representations of the character to be used by the collision system. The idea is that the coarse-level representation allows for sufficient accuracy and speed in the collision system but is not detailed enough for visual purposes. The automatic generation of levels of detail in many of the current algorithms allows us to create the coarse resolution representations for collision detection purposes.

This chapter is by no means a detailed description of all the various ideas and algorithms developed over the past few years. It is intended to give you a flavor of

the concepts that any game program must handle when incorporating level of detail. The simplest form of level of detail involves 2D representations of 3D objects, called *sprites* or *billboards*, and is discussed in Section 7.1. Switching between models of varying degrees of resolution, a process called *discrete level of detail*, is a step up in quality from 2D representations. The switching is usually associated with distance from eye point to object. Section 7.2 covers the basic concepts for this topic. A form of geometric control for visual quality called *continuous level of detail* is discussed in Section 7.3, with a brief discussion of how it applies to models and to terrain. Section 7.4 is about *infinite level of detail*, which is essentially what you have for surface-based models. As long as you have the cycles to burn, you can tessellate a surface to whatever level of detail you like and feed the triangles to the rendering system. The topic of geometric level of detail is quite broad. A good book covering many algorithms and issues is [LRC⁺03].

Based on what I've gleaned from some of the game developer forums, continuous level of detail—in particular for terrains—has lost some of its popularity. The video memory capacity of consumer graphics cards and of game consoles has grown enough and the GPUs have sufficient rendering throughput for you to process detailed models and terrain at their original high resolution. However, the concepts are still important for very large terrains and for automatic generation of low-resolution models for the purpose of collision detection.

7.1 SPRITES AND BILLBOARDS

The simplest form of level of detail uses *sprites*, sometimes called *impostors*. These are pre-rendered images of 3D objects. The idea is that the time it takes to draw the image as a texture is much shorter than the time required to render the object. In a 3D environment, sprites are useful for software rendering simply because of the reduction in the time to draw. However, sprites are usually easy to spot in a rendering if they represent objects that are close to the eye point or if the eye point moves. The image gives the impression that the object is not changing correctly with eye point location or orientation. The visual anomaly due to closeness to eye point is softened if sprites are only used for distant objects, for example, trees drawn in the distance.

The visual anomaly associated with a moving eye point can be rectified in two ways. The first way is to have a set of pre-rendered images of the object calculated from a set of predefined eye points and orientations. During application execution, a function is used to select the appropriate image to draw based on the current location of the eye point. The second way is to allow a single pre-rendered image to change orientation depending on eye point location and orientation. In this setting the sprite is called a *billboard*.

Billboards can change orientation based on a few schemes. All calculations are assumed to be in the model space of the billboard. During application execution, the eye point and orientation vectors are transformed from world space to the model

space of the billboard, and the billboard's new alignment is calculated. The basic billboard consists of a rectangle (two triangles) and a textured image. A coordinate system is assigned to the billboard: its origin is the center point of the rectangle; the edge directions are two coordinate axes; and the normal to the plane of the rectangle is the third coordinate axis. A billboard can be *screen aligned*. The billboard is first rotated so that its normal vector is aligned with the view direction. Within this new plane, the billboard is rotated so that its model-space up vector is aligned with the view up vector. Screen alignment is good for displaying particles in a particle system and for isotropic textures such as smoke clouds. If the texture is anisotropic (e.g., a tree texture), then screen alignment does not make sense in the case when the viewer rotates about the current view direction. The tree should remain upright even though the viewer is tilting his or her head. For these types of billboards, *axial alignment* is used. The billboard is allowed to rotate only about its model-space up vector. For a given eye point, the billboard is rotated so that its normal vector is aligned with the vector from the eye point to its projection onto the up axis of the billboard.

Note that the alignment of a billboard relative to an eye point requires identifying a coordinate frame for the billboard and changing that frame with respect to the eye point's coordinate frame. The idea of alignment can therefore be extended to a fully 3D object as long as a coordinate frame is assigned to that object. In this sense, a billboard class can be defined for a special type of node in a scene graph, and the children of that node can be arbitrary objects, not just flat polygons and images.

7.2 DISCRETE LEVEL OF DETAIL

A simple LOD algorithm is to construct a sequence of models whose triangle count diminishes over the sequence. The sequence is assigned a center point that is used as a representation of the centers for all the models. The model with the largest number of triangles is drawn when the LOD center for that model is close to the camera. As the center point moves farther away from the camera, and at some preselected distance, the current model is replaced by the next model in the sequence. To support this "switch," the hierarchical scene graph has a node type designated as a *switch node*. This node type provides an interface that allows the application to select which child of the node should be processed in any recursive traversals of the scene graph. Only one child may be active at a time. The scene graph can then support specialized switch nodes, one of those being an *LOD node*. The children of an LOD node are the models in the sequence. The node itself maintains the center point. During a rendering pass when the LOD node is visited, its pre-render function computes the distance from the center point to the eye point and sets the appropriate active child to propagate the rendering call.

The word *discrete* refers to the fact that the number of models is a small finite number. The advantage of discrete level of detail is the simplicity of the implementation. The disadvantage is that an artist must build all those models. Moreover,

whenever a switch occurs during rendering, it is usually noticeable and not very natural—the *popping* effect. One approach that has been taken to reduce the popping is to morph between two consecutive models in the LOD sequence. This requires establishing a correspondence between the vertices of the models and is problematic when the number of vertices is different between the two. The morphing is implemented as convex combinations of paired vertices, with the weighting factors dependent on the switching distances for the models. That is, if d_1 is the distance at which model 1 switches to model 2, and if d_2 is the distance at which model 2 switches to model 3, while the LOD center is a distance $d \in [d_1, d_2]$ from the eye point, the weight $(d - d_1)/(d_2 - d_1)$ is applied to vertices in model 1 and the weight $(d_2 - d)/(d_2 - d_1)$ is applied to vertices in model 2. The results might be acceptable, but the price to be paid for each frame makes for an expensive interpolation. However, the results might not be visually appealing, since the morphing is not based on preserving geometric information about the models. The quality of the end result depends a lot on the quality and differences in the original models.

7.3 CONTINUOUS LEVEL OF DETAIL

An alternative to discrete level of detail is *continuous level of detail* (CLOD). One major category of CLOD algorithms includes progressive meshes that simplify already existing triangle meshes [Hop96a, Hop96b, GH97, GH98, CVM⁺96, COM98, LE97, LT98]. Some of the later papers realized the importance of also simplifying the surface attributes (e.g., vertex colors and texture coordinates) in a visually appealing way. The basic concept is one of triangle reduction. The Garland-Heckbert algorithm [GH97] is particularly well suited for a game engine and is discussed in this section. This algorithm effectively builds a large sequence of models from the highest-resolution model, so in a sense it is like discrete LOD, but it does not require an artist to build the additional models. The change in the number of triangles between consecutive models is a small number, so popping is not as noticeable, particularly when a screen-space error metric is used to control the triangle changes rather than the distance of the model center from the eye point.

7.3.1 SIMPLIFICATION USING QUADRIC ERROR METRICS

The Garland-Heckbert algorithm [GH97] creates a sequence of incremental changes to the triangle mesh of the original model by contracting pairs of vertices in a way that attempts to preserve geometric information about the model rather than topological information. Other researchers have used methods that typically require manifold topology and do not necessarily handle shape in a reasonable way. Vertex decimation involves removing a vertex and all triangles sharing it, then retriangulating the hole that was produced by removal [SZL92]. Vertex clustering involves placing the mesh in a bounding box, partitioning that box into a lattice of small boxes, collapsing all

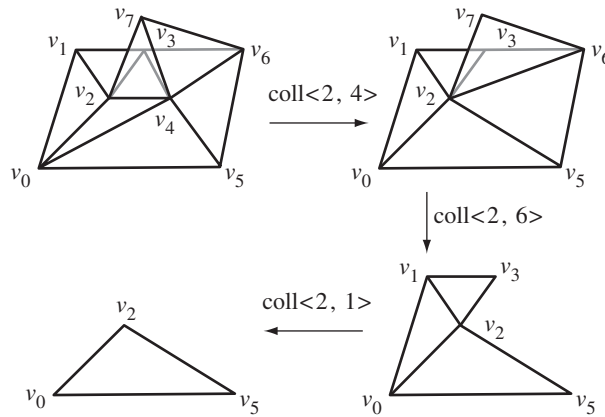


Figure 7.1 A sequence of three edge collapses in a triangle mesh.

vertices in each small box into a single vertex, and removing and adjusting the triangles of the original mesh accordingly [RB93]. Iterative edge contraction involves replacing an edge and its two vertices by a single vertex, removing the triangles sharing that edge, and adjusting the connectivity information for the triangles adjacent to the ones removed [Hop96a].

The Garland-Heckbert algorithm is based on iterative vertex contraction and is not restricted to a manifold topology. Moreover, two disjoint components of a triangle mesh might very well be joined by this algorithm, so mesh topology is not necessarily preserved. This is not a drawback to the algorithm, because a mesh that appears as two distinct objects while close to the eye point might look like a single object while in the distance. Merging of components by the simplification algorithm supports this. The basic contraction involves a vertex pair (V_1, V_2) that is replaced by a single vertex V . The original vertices are in a sense moved to the new vertex, V_1 becomes V , and V_2 is removed. The edges that shared V_2 are now connected to V , and any edges or faces that become degenerate are removed. Figure 7.1 illustrates the contraction of three pairs of vertices (three edges). In fact, the contraction process can be applied to an entire set of vertices $\{V_i\}_{i=1}^m \rightarrow V$ if desired. Simplification amounts to taking the original mesh M_0 and creating a sequence of n vertex contractions to produce a sequence of meshes $M_0, M_1, \dots, M_n$.

The algorithm requires identification of those vertices in the original mesh that can be contracted. A pair (V_1, V_2) is said to be a *valid pair* for contraction if the two points are endpoints of the same edge or if $|V_1 - V_2| < \tau$ for some threshold parameter $\tau > 0$ specified by the application. If $\tau = 0$, then the vertex contraction

is really an edge contraction. Positive thresholds allow nonconnected vertices to be paired.

The algorithm also requires taking the set of valid pairs and associating with each pair a metric that is used to prioritize the pairs. The smaller the metric, the more likely the pair should be contracted. This is accomplished by associating with each vertex $\mathbf{V} = (\mathbf{X}, 1)$, treated as a homogeneous vector, a symmetric 4×4 matrix $Q(\mathbf{V})$, and choosing the metric to be the quadratic form

$$E(\mathbf{X}) = \mathbf{V}^T Q \mathbf{V} = \begin{bmatrix} \mathbf{X}^T & 1 \end{bmatrix} \left[\begin{array}{c|c} A & \mathbf{B} \\ \hline \mathbf{B}^T & c \end{array} \right] \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} = \mathbf{X}^T A \mathbf{X} + 2\mathbf{B}^T \mathbf{X} + c$$

where A is a 3×3 symmetric matrix, $\mathbf{B}$ is a 3×1 vector, and c is a scalar. Note that $E(\mathbf{X}) = d$ for a constant d defines a quadric surface. A specific matrix Q is constructed in the next section, but for the purpose of simplification it could be one of many choices.

Given a valid pair $(\mathbf{V}_1, \mathbf{V}_2)$, there are two things to do. The first thing to do is to compute the target $\mathbf{V}$ of the contraction. While simple choices are $\mathbf{V}_1$ or $\mathbf{V}_2$ (replacement) or $(\mathbf{V}_1 + \mathbf{V}_2)/2$ (averaging), a better choice is to choose $\mathbf{V}$ so that $E(\mathbf{X})$ is minimized. This occurs when $\nabla E(\mathbf{X}) = \mathbf{0}$, which leads to solving $A\mathbf{X} = -\mathbf{B}$. If A is invertible, then the solution $\mathbf{X}$ is used to generate the contracted vertex $\mathbf{V} = (\mathbf{X}, 1)$. However, if A is not invertible, then the minimization problem is restricted to the line segment $\mathbf{X}(t) = \mathbf{X}_0 + t(\mathbf{X}_1 - \mathbf{X}_0)$ for $t \in [0, 1]$. The function to minimize is the quadratic in one variable, $\phi(t) = E(\mathbf{X}(t))$. The minimum occurs either where $\phi'(t) = 0$ with $t \in [0, 1]$ or at an endpoint $t = 0$ or $t = 1$. The second thing to do is associate a metric with $\mathbf{V}$. A simple choice is $Q = Q_1 + Q_2$, where Q_i is the metric for $\mathbf{V}_i$, $i = 1, 2$.

The Algorithm

A model is represented using a vertex-edge-face table to store the connectivity information. Each vertex keeps track of a list of other vertices to which it is adjacent. The algorithm is as follows:

1. Compute Q for all vertices.
2. Compute all valid pairs based on a selected $\tau \geq 0$.
3. Compute $\mathbf{V}$ for each pair $(\mathbf{V}_1, \mathbf{V}_2)$, $Q = Q_1 + Q_2$, and $\mathbf{V}^T Q \mathbf{V}$.
4. Place all pairs in a heap whose first element is that pair with minimum $\mathbf{V}^T Q \mathbf{V}$ for the vertex $\mathbf{V}$ that is contracted from the pair.
5. The first pair in the list is contracted to form the new mesh and is removed from the heap. The valid pairs affected by the removal have their metrics recalculated. The pair with the minimum error term is moved to the front of the heap, and this step is repeated until the heap is empty.

A couple of potential problems need to be dealt with. The first problem is that the algorithm does not handle open boundaries in any special way. For some models (such as terrain) it might be important to tack down the boundary edges of the mesh. One way to do this is to generate a plane through each boundary edge that is perpendicular to the triangle containing that edge. The quadric matrix is calculated, weighted by a large penalty factor, and added into the quadric matrices for the endpoints of the edge. It is still possible for boundary edge vertices to move, but it is highly unlikely. Another way is never to allow a boundary edge vertex to be moved or replaced in the simplification. The implementation on the CD-ROM locates the boundary edges and assigns infinite weights to them, so the boundary edges can never be removed by an edge collapse.

The second problem is that pair contractions might not preserve the orientation of the faces near the contraction, so a folding over of the mesh occurs. A method to prevent this is to compare the normal vector of each neighboring face before and after the contraction. If the normal vector changes too much, the contraction is disallowed.

Construction of the Error Metric

A heuristic is chosen for the quadric error metric. Each vertex in the mesh is in the intersection of the planes containing the triangles that share that point. If a plane is represented as $\mathbf{P}^T \mathbf{V} = 0$, where $\mathbf{V} = (\mathbf{X}, 1)$ and $\mathbf{P} = (\mathbf{N}, d)$ with $|\mathbf{N}| = 1$, define $S(\mathbf{V})$ to be the set of vectors $\mathbf{P}$ representing the planes containing the triangles that share $\mathbf{V}$. The error of $\mathbf{V}$ with respect to $S(\mathbf{V})$ is the sum of squared distances from $\mathbf{V}$ to its planes:

$$E(\mathbf{V}) = \sum_{\mathbf{P} \in S(\mathbf{V})} (\mathbf{P}^T \mathbf{V})^2 = \mathbf{V}^T \left(\sum_{\mathbf{P} \in S(\mathbf{V})} \mathbf{P} \mathbf{P}^T \right) \mathbf{V} = \mathbf{V}^T \mathbf{Q}(\mathbf{V}) \mathbf{V}$$

where the last equality defines $\mathbf{Q}(\mathbf{V})$ for a given vertex. The matrix $\mathbf{P} \mathbf{P}^T$ is called a fundamental error quadric and, when applied to any point $\mathbf{W}$, measures the squared distance from that point to the plane.

The initial vertices have matrix $\mathbf{Q}(\mathbf{V}) \neq 0$, but the initial error estimates are $\mathbf{V}^T \mathbf{Q}(\mathbf{V}) \mathbf{V} = 0$. On the first iteration of the algorithm, the sum of two quadric error matrices will generate another nonzero quadric error matrix whose quadratic form usually has a positive minimum.

Topological Considerations

An example that illustrates how a mesh can fold over, independently of the geometry of the mesh, is shown in Figure 7.2. In part (a) of the figure, the triangles are counterclockwise ordered as $\langle 0, 4, 3 \rangle$, $\langle 4, 1, 2 \rangle$, and $\langle 4, 2, 3 \rangle$. The collapse of vertex

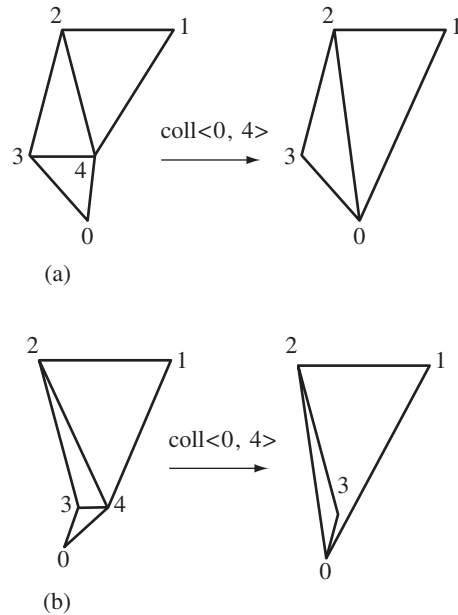


Figure 7.2 An edge collapse resulting in the mesh folding over on itself: (a) no folding and (b) folding.

4 to vertex 0 leads to deletion of $\langle 0, 4, 3 \rangle$, modification of $\langle 4, 1, 2 \rangle$ to $\langle 0, 1, 2 \rangle$, and modification of $\langle 4, 2, 3 \rangle$ to $\langle 0, 2, 3 \rangle$. Both modified triangles are visible in the figure as counterclockwise.

In part (b) of the figure, the modified triangle $\langle 0, 2, 3 \rangle$ is counterclockwise. This is by design; collapses always preserve this. But the triangle appears to be clockwise in the figure: upside down, it folded over. We can avoid the problem by doing a look-ahead on the collapse. If any potentially modified triangle causes a folding, we assign an infinite weight to the offending edge to prevent that edge from collapsing.

Another issue when collapsing edges in an open mesh is that the mesh can shrink. To avoid shrinking, we can also assign infinite weights to boundary edges of the original mesh. And, finally, if we want to preserve the mesh topology, we can assign infinite weights to edges with three or more shared triangles.

Simplification at Run Time

A class `ClodMesh` can be derived from `Spatial` and added to the collection of geometric-type leaf nodes that can be placed in a scene graph and rendered. A

ClodMesh object represents the mesh sequence M_0 through M_n , where M_0 is highest resolution. The original mesh is assumed to have manifold topology and the simplification is assumed to do edge contractions. While neither of these is a requirement of the algorithm as published, they do make the implementation a bit more manageable. A consequence of the two assumptions is that two consecutive meshes in the simplification differ by one or two triangles. Moreover, the sequence is assumed to be precomputed, thereby gaining execution speed at the cost of memory usage.

An automated selection can be made at each frame to display mesh M_i for some appropriate index i by using the pre-rendering virtual function. Although there are many possibilities for selection, a simple one uses screen-space coverage by the world bounding volume containing mesh M_0 . If A is the screen-space area covered by the bounding volume and if τ is an application-specified number of triangles per pixel, then the number of requested triangles in the mesh to be drawn is $A\tau$. This number is clamped to $[T_0, T_n]$, where T_i is the number of triangles in mesh M_i . The index j is chosen so that $T_j = \lfloor A\tau \rfloor$, and the mesh M_j is identified as the one to be drawn in the rendering virtual function.

Selecting Surface Attributes

If the original mesh has surface attributes at the vertices such as normal vectors, texture coordinates, and colors, then new surface attributes must be selected for a contracted vertex. For a single edge contraction, it is reasonable to select texture coordinates and colors based on the values at the vertices of the two triangles sharing that edge. If the two triangles are not coplanar, then the four vertices form a tetrahedron. A simple scheme to compute a new scalar value based on four old ones is to compute the barycentric coordinates of the new vertex with respect to the tetrahedron and use them in a weighted average of the scalar values. Some care must be taken if the new vertex is not inside the tetrahedron so that at least one of the barycentric coordinates is outside the interval $[0, 1]$. To remedy this, any negative barycentric coordinates are clamped to 0 and the coordinates are rescaled to sum to 1. The resulting coordinates are used to compute a convex combination of the four scalar values. Normal vectors may be recomputed at vertices for which the shared triangle set has changed.

7.3.2 REORDERING OF VERTICES AND INDICES

The decimation scheme here produces a sequence of meshes, the first mesh the original data and each additional mesh a lower-resolution version of the previous one. The vertices and indices of the original mesh can be reordered so that all meshes use the vertex buffer and index buffer of the original mesh. Each mesh stores the number of vertices and number of indices for the subsets of the buffers it needs.

Think about why this works by considering a single edge collapse. If the collapse causes vertices V_{i_1} through V_{i_n} to be removed from the mesh, you can move these

vertices to the end of the original vertex buffer. Essentially, this is a permutation of the vertex indices. Moving the vertices causes the indices in the index buffer to be invalid, but you can apply the permutation to reassign the indices and obtain a valid index array. The collapse also causes triangles T_{j_1} through T_{j_m} to be removed from the mesh. You can move the indices to the end of the original index buffer. Since it really does not matter in which order you draw the triangles, this rearrangement is okay to do.

On the next edge collapse, the vertices that are removed by the collapse are moved to the vertex buffer location just before those vertices that were moved because of the previous collapse. The same idea applies to the indices.

By reordering the vertices and indices, you may transfer the vertex buffer and index buffer of the original mesh to video memory *once*. Drawing of lower-resolution meshes amounts to telling the graphics system to enable these buffers, but also telling it the different vertex and index quantities to use. Without the reordering, you would have to send vertex and index buffers across the bus on each frame, which is slow.

7.3.3 TERRAIN

In the first edition of the book, I had described in detail a continuous-level-of-detail algorithm for height fields [LKR⁺96]. We had implemented this in NetImmerse with modifications to improve the performance. Later, another paper appeared with an algorithm that became more popular—the ROAM algorithm [DWS⁺97], but I had no time to investigate this when finishing up the first edition. The goal of both CLOD algorithms is to reduce the number of triangles that the renderer must draw. In exchange for fewer triangles, you spend CPU cycles trying to decide which triangles to feed to the renderer. There is a fine balance in performance here. Spend too much time on the decision process? You slow down the application and starve the renderer. Spend too little time on the decision process, perhaps with a more conservative estimate, and you slow down the renderer because it has to process too many triangles. Moreover, visual quality is important in that you do not want triangles to pop in and out of view as the camera moves. This argues for more CPU time because you need to use screen-space metrics to decide on the number of triangles.

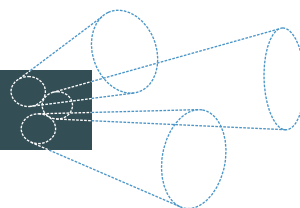
The CLOD terrain algorithms were important because, at the time, the amount of memory on hardware-accelerated cards was sufficiently small that you had to send terrain data over the bus to video memory on each frame. Reducing the triangle count meant reducing the bus traffic. Current-generation graphics hardware has much more video memory. My experience lately has been to use just regular terrain pages without any CLOD algorithm applied to them. The pages are all loaded in video memory once. As the camera moves, eventually you reach a point where you need to load new pages for terrain about ready to be encountered, and you can discard current pages that are no longer visible. It is quite simple to manage terrain pages,

loading and discarding using simple heuristics based on camera position, orientation, and speed.

Programmable graphics hardware is powerful enough now that other algorithms are appearing in the literature, using the GPUs to handle continuous level of detail. A popular algorithm uses *geometry clipmaps* as presented in the paper [LH04] and appearing in a chapter of the book [Pha05]. Other CLOD algorithms are referenced at the Virtual Terrain Project website [Pro06].

7.4 INFINITE LEVEL OF DETAIL

Another major category of CLOD algorithms includes the dynamic tessellation of surfaces that are defined functionally. Because there is no theoretical bound on the number or size of triangles that can be created in the tessellation, this type of algorithm provides an infinite level of detail. Of course, there is a practical bound based on the number of triangles an engine can process to maintain a high frame rate and the amount of memory available. The benefit, though, is compactness of representation of the model. See Section 12.7 for a discussion of dynamic tessellation.



COLLISION DETECTION

Collision detection is a very broad topic, relevant to computer games and to other applications such as navigation and robotics. The classic example for collision detection in a third-person perspective, indoor game is having the main character move around in a set of rooms that contain obstacles. The character is controlled by an input device, typically a joystick, keyboard, or mouse, and must not be allowed to walk through the walls or obstacles. Moreover, if the character walks into a wall, he might be allowed to slide along the wall in a direction that is oblique to the one implied by the event from the input device. A standard technique for preventing the character from walking through a wall is to enclose the character with a tight-fitting bounding volume and test whether it intersects the plane of the wall. The collision detection system must provide support for this test even when the character (and its bounding volume) is moving. Preventing the character from walking through an obstacle is as simple as enclosing the obstacle with its own bounding volume and testing for intersection between the character and obstacle bounding volumes. Other typical situations in a game that require collision detection are keeping vehicles moving over a terrain without dropping through it, monitoring racing cars on a track and detecting when two cars hit or when a car hits a wall, determining when a projectile hits an intended target, bouncing objects off other objects, providing feedback about character control when two characters are fighting, and determining if an object can pass through an opening, such as when a character attempts to walk through a doorway that may or may not be tall enough.

The nature of collision detection algorithms depends on the types of objects involved and what information is needed by the caller of a collision detection query. I choose to categorize collision detection algorithms according to the following broad categories:

1. *Stationary objects.* Both objects are not moving.
 - (a) *Test-intersection queries.* Determine if the two objects are intersecting. The algorithms need only determine if the objects are intersecting, not what the set of intersection is.
 - (b) *Find-intersection queries.* Determine the intersection set of the objects. This set is the empty set when the objects are not intersecting.
2. *Moving objects.* One or both objects are moving. If both are moving, you may subtract the velocity of the first from the velocity of the second and handle the problem as if one object were stationary and the other were moving. Invariably, the application will limit the time interval over which the intersection query applies, say, $[0, t_{\max}]$ for some user-specified $t_{\max} > 0$. If the objects intersect during that time interval, they will intersect at a first time $t_{\text{first}} \in [0, t_{\max}]$, called the *contact time*. The set of intersection at the first time is called the *contact set*.
 - (a) *Test-intersection queries.* Determine if the two objects will intersect during the time interval. The algorithms need only determine if the objects will intersect. The contact time might not be needed by the application. Since it is a natural consequence of the intersection algorithm, it is usually returned to the caller anyway. The query does not involve computing the contact set.
 - (b) *Find-intersection queries.* Determine the contact time and contact set of the objects. This set is the empty set when the objects do not intersect during the time interval.

As you can see, even a general categorization leads to a lot of possibilities that a collision detection system must handle.

At the lowest level you must decide whether to use a *distance-based method* or an *intersection-based method*. A distance-based method is usually implemented by choosing a parametric representation of the objects. A quadratic function of the parameters is constructed and represents the squared distance between pairs of points, one point per object. A constrained minimization is applied to the quadratic function to find the pair of closest points. An intersection-based method is also usually implemented by choosing parametric representations for the two objects, equating them, and then solving for those parameters. Distance algorithms for commonly encountered objects are discussed in Chapter 14. Intersection algorithms for these objects are discussed in Chapter 15.

Intersection-based methods are generally simpler to design and implement than distance-based methods because they use only basic algebra. The distance-based approach is quite a bit more complicated and uses calculus. Why would you ever consider using the distance-based approach? For a pair of objects such as a line and a plane? Never. However, as the complexity of the object representations increases, the algebraic details for equating parametric representations and solving become greater

and more tedious to implement. Naturally, those same representations are used in the distance-based methods, but iterative algorithms may be used to compute the closest pair of points rather than exact algorithms to produce closed-form equations for the parameters corresponding to the closest pair of points.

Consider the case of moving objects for which you want to know the first time of contact. The design and implementation of intersection-based algorithms can be difficult, depending on the type of objects. Moreover, you might try a bisection algorithm on the time interval of interest, $[0, t_{\max}]$. If the objects are not intersecting at time 0, but they are intersecting at time $t_{\max}$, you may test for an intersection at time $t_{\max}/2$. If the objects are intersecting at this time, you repeat the test at time $t_{\max}/4$. If the objects are not intersecting at this time, you repeat the test at time $3t_{\max}/4$. The subdivision of the time interval is repeated until you reach a maximum number of subdivisions (provided by the application) or until the width of the current time subinterval is small enough (threshold provided by the application). The fact that the subdivision is guided solely by the Boolean results at the time interval endpoints (intersecting or not intersecting) does not help you formulate a smarter search for the first contact time. A distance-based algorithm, on the other hand, gives you an idea of *how close* you are at any specified time, and this information supports a smarter search.

Even an iterative algorithm based on distance can be complicated to formulate and implement because of the complicated nature of the distance functions themselves. As it turns out, there is a good compromise that has the flavor of both intersection- and distance-based methods. The idea is to use a *pseudodistance* function for the two objects. This function is positive when the objects are separated, negative when the objects are overlapping, and zero when the objects are just touching. Moreover, the magnitude of the function is approximately proportional to the actual distance between the objects when separated and to the size of the common region when overlapping. Pseudodistance functions are generally easier to formulate and evaluate than distance functions. Some of these are based on measuring the separation or overlap of objects. A popular method for such measurements is the *method of separating axes*, the topic of Section 8.1. Section 8.2 shows how to design a pseudodistance-based iterative method for determining the first time of contact between two objects. The system is general, requiring you minimally to implement a pseudodistance function for each pair of object types of interest. The system is also extensible, allowing you to override many of the generic functions when you want to take advantage of the special structure of your objects. Section 8.3 covers the details of building the general system.

Intersection and distance algorithms for general objects can be extremely complicated. For that reason, practitioners restrict their attention to what are called *convex objects*. If S is a set of points representing the object, the set is said to be *convex* whenever the line segment connecting two points $\mathbf{X}$ and $\mathbf{Y}$ is contained in the set, no matter which two points you choose. That is, if $\mathbf{X} \in S$ and $\mathbf{Y} \in S$, then $(1 - t)\mathbf{X} + t\mathbf{Y} \in S$ for all $t \in [0, 1]$. Figure 8.1 shows two planar objects, one convex and one not convex.

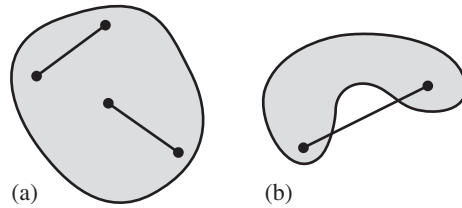


Figure 8.1 (a) A convex object. No matter which two points you choose in the set, the line segment connecting them is in the set. (b) A nonconvex object. A pair of points in the set is shown for which the line segment connecting the points is not fully in the set.

For objects that are not convex, a typical approach to computing intersection or distance is to decompose the object into a union of convex sub-objects (not necessarily disjoint) and apply the intersection-based and/or distance-based queries to pairs of convex sub-objects, with one sub-object from each object in the intersection query.

Implementing a robust collision detection system is a difficult and elusive task, as many game programmers have found. The algorithms for *dynamic* (moving) objects tend to be somewhat more difficult to implement than for *static* (nonmoving) objects, particularly because of the implied increase in dimension (four dimensions, three in space and one in time). A very good book, which is entirely on real-time collision detection and covers many topics, is [Eri04]. A more specialized book is [vdB03], which describes a system that computes distance between convex polyhedra using the GJK algorithm [GJK88, GF90, Cam97, vdB99].

A couple of topics that are collision related are covered in the final two sections of the chapter. These are not usually considered part of a collision detection system that you would find in a physics engine. Section 8.4 is on the topic of *picking*. If you want to use the mouse to click on an object and interact with it, you can do so by constructing a linear component in world coordinates that corresponds to the selected pixel, and then applying an intersection query between that linear component and objects in the scene. Line-object intersection queries may also be used for *collision avoidance*. In order to move the camera and not walk through walls or other objects, you can generate a small number of lines in various directions, test if any of these intersect objects within a small distance from the camera, and then allow the camera to move if there will be no intersections. A more complicated system for collision avoidance uses preprocessed information about your environment. This information is used for *pathfinding*, the process of selecting a starting point and an ending point, and then determining an unobstructed path between the two. This is the topic of Section 8.5.

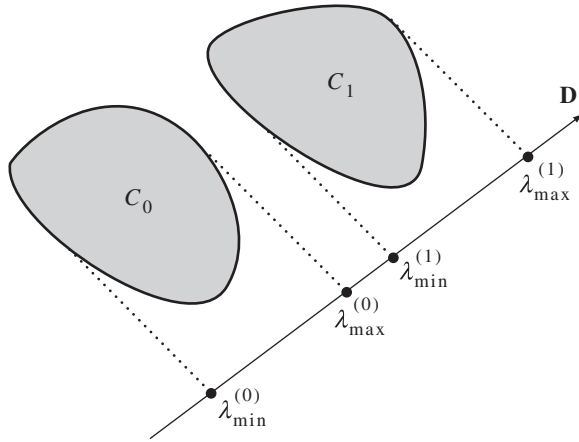


Figure 8.2 Nonintersecting convex objects and a separating line for them. The algebraic condition for separation is $\lambda_{\max}^{(0)}(\mathbf{D}) < \lambda_{\min}^{(1)}(\mathbf{D})$ as indicated in Equation (8.1).

8.1 THE METHOD OF SEPARATING AXES

A test for nonintersection of two convex objects is simply stated: If there exists a line for which the intervals of projection of the two objects onto that line do not intersect, then the objects do not intersect. Such a line is called a *separating line* or, more commonly, a *separating axis*. Figure 8.2 illustrates.

The translation of a separating line is also a separating line, so it is sufficient to consider lines that contain the origin $\mathbf{O}$ and with unit-length direction $\mathbf{D}$, the projection of a convex set C onto the line is the interval

$$I = [\lambda_{\min}(\mathbf{D}), \lambda_{\max}(\mathbf{D})] = [\min\{\mathbf{D} \cdot (\mathbf{X} - \mathbf{O}) : \mathbf{X} \in C\}, \max\{\mathbf{D} \cdot (\mathbf{X} - \mathbf{O}) : \mathbf{X} \in C\}]$$

where possibly $\lambda_{\min}(\mathbf{D}) = -\infty$ or $\lambda_{\max}(\mathbf{D}) = +\infty$, these cases arising when the convex set is unbounded. Two convex sets C_0 and C_1 are *separated* if there exists a direction $\mathbf{D}$ such that the projection intervals I_0 and I_1 do not intersect. Specifically, they do not intersect when

$$\lambda_{\min}^{(0)}(\mathbf{D}) > \lambda_{\max}^{(1)}(\mathbf{D}) \quad \text{or} \quad \lambda_{\max}^{(0)}(\mathbf{D}) < \lambda_{\min}^{(1)}(\mathbf{D}) \quad (8.1)$$

The superscript corresponds to the index of the convex set. Although the comparisons are made where $\mathbf{D}$ is unit length, the comparison results are invariant to changes in length of the vector. This follows from $\lambda_{\min}(t\mathbf{D}) = t\lambda_{\min}(\mathbf{D})$ and $\lambda_{\max}(t\mathbf{D}) = t\lambda_{\max}(\mathbf{D})$ for $t > 0$. The Boolean value of the pair of comparisons is also invariant

when $\mathbf{D}$ is replaced by the opposite direction $-\mathbf{D}$. This follows from $\lambda_{\min}(-\mathbf{D}) = -\lambda_{\max}(\mathbf{D})$ and $\lambda_{\max}(-\mathbf{D}) = -\lambda_{\min}(\mathbf{D})$. When $\mathbf{D}$ is not unit length, the intervals obtained for the separating line tests are not the projections of the object onto the line; rather, they are scaled versions of the projection intervals. We make no distinction between the scaled projection and regular projection. We will also use the terminology that the direction vector for a separating line is called a *separating direction*, a direction that is not necessarily unit length.

Please note that in two dimensions, the terminology of a separating line or axis is potentially confusing. The separating line separates the *projections* of the objects on that line. The separating line does *not* partition the plane into two regions, each containing an object. In three dimensions, the terminology should not be confusing since a plane would need to be specified to partition space into two regions, each containing an object. No real sense can be made for partitioning space by a line.

8.1.1 EXTREMA OF CONVEX POLYGONS OR CONVEX POLYHEDRA

Since the method of separating axes involves testing for separation of projection intervals, we need to know how to compute these intervals in the first place. The projection interval of a convex polygon or convex polyhedra may be computed by projecting all the vertices of the object and selecting the extreme values. A simple algorithm for computing an extreme vertex for a convex polygon or polyhedron with vertices $\mathbf{P}_i$ for $0 \leq i < n$ in the direction $\mathbf{D}$ is to locate the vertex $\mathbf{P}_j$ for which $\mathbf{D} \cdot \mathbf{P}_j = \max_{0 \leq i < n} \{\mathbf{D} \cdot \mathbf{P}_i\}$. This is clearly an $O(n)$ algorithm. The question is, Can we find an algorithm that is asymptotically more efficient, say, $O(\log n)$? An $O(\log n)$ algorithm will outperform an $O(n)$ algorithm *in the limit as n approaches infinity*, but it is possible that an $O(n)$ algorithm outperforms an $O(\log n)$ algorithm for small- or medium-sized n . In practice, it is important to have some measurements of the constant in the asymptotic order. Moreover, given implementations of the competing $O(n)$ and $O(\log n)$ algorithms, it is worthwhile to determine the *break-even* value of n —the value at which the $O(\log n)$ algorithm outperforms the $O(n)$ algorithm.

An extremal query for a convex polygon can be performed in $O(\log n)$ with no preprocessing of the polygon other than guaranteeing that its vertices are ordered. The algorithm is effectively a bisection of the dot products $\mathbf{D} \cdot \mathbf{P}_i$. This method does not have a counterpart for convex polyhedra. As it turns out, an extremal query algorithm of $O(\log n)$ does exist for convex polyhedra, but it requires a data structure that takes $O(n)$ time to build. For applications that have repetitive queries, the preprocessing time is not important. The idea for the data structure is due to [DK90] and [Kir83], and it is referred to as the *Dobkin-Kirkpatrick hierarchy*. The algorithm itself, both the construction of the data structure and the extremal query, are discussed in detail in a well-written chapter in [O'R98]. The construction relies on finding maximum independent sets in graphs, a problem known to be NP-complete. However,

[EM85, Ede87] provide an approximation that gives sufficiently large independent sets that lead to an $O(n)$ construction while maintaining $O(\log n)$ for the query.

The construction is quite elegant and the details provided in [O'R98] are enough to get you started on implementing the algorithm. Even so, the algorithm is intricate, requires some high-powered machinery to implement, including convex hull construction, and makes an implementation a formidable challenge. In this section, I provide an alternative to the Dobkin-Kirkpatrick hierarchy. It is based on constructing a BSP tree for the spherical dual of a convex polyhedron. The BSP tree construction is $O(n)$ and the extremal query is $O(\log n)$ as long as you have a reasonably balanced tree. A heuristic for creating balanced trees is provided here. The implementation of the query is trivial and requires only a few lines of code. The construction of the tree is more complicated and assumes that a specific graph data structure exists for representing adjacency information for the vertices, edges, and triangles of the convex polyhedron.

Extremal Query for a Convex Polygon

Consider a convex polygon with counterclockwise-ordered vertices $\mathbf{P}_i$ for $0 \leq i < n$. The edge directions are $\mathbf{E}_i = \mathbf{P}_{i+1} - \mathbf{P}_i$, where it is assumed we are using modular arithmetic on the indices for wraparound, $\mathbf{P}_n = \mathbf{P}_0$ and $\mathbf{P}_{-1} = \mathbf{P}_{n-1}$. Outward-pointing, unit-length normals $\mathbf{N}_i$ may be constructed for the edges. The normal vectors may be drawn as points on a unit circle. The arcs connecting the points correspond to the edges of the polygon. This view of the circle is called the *polar dual* of the polygon. Figure 8.3 illustrates for a six-sided polygon.

If $\mathbf{D} = \mathbf{N}_i$, then all points on the edge $\mathbf{E}_i$ are extremal. If $\mathbf{D}$ is strictly between $\mathbf{N}_0$ and $\mathbf{N}_1$, then $\mathbf{P}_1$ is the unique extremal point in that direction. Similar arguments apply for $\mathbf{D}$ strictly between any pair of consecutive normals. The normal points on the circle decompose the circle into arcs, each arc corresponding to an extremal vertex of the polygon. An endpoint of an arc corresponds to an entire edge being extremal. The testing of $\mathbf{D}$ to determine the full set of extremal points is listed below, where $(x, y)^\perp = (y, -x)$:

- Vertex $\mathbf{P}_i$ is optimal when $\mathbf{N}_{i-1} \cdot \mathbf{D}^\perp > 0$ ($\mathbf{D}$ is *left* of $\mathbf{N}_{i-1}$) and $\mathbf{N}_i \cdot \mathbf{D}^\perp < 0$ ($\mathbf{D}$ is *right* of $\mathbf{N}_i$).
- Edge $\mathbf{E}_i$ is optimal when $\mathbf{N}_{i-1} \cdot \mathbf{D}^\perp = 0$ and $\mathbf{N}_i \cdot \mathbf{D}^\perp < 0$.

The indexing is computed in the modular sense, where $\mathbf{N}_n = \mathbf{N}_0$ and $\mathbf{N}_{-1} = \mathbf{N}_{n-1}$. Assuming we will be projecting the extremal set onto the query axis, we can collapse the two tests into a single test and just use one vertex of an extremal edge as the to-be-projected point:

- Vertex $\mathbf{P}_i$ is optimal when $\mathbf{N}_{i-1} \cdot \mathbf{D}^\perp \geq 0$ and $\mathbf{N}_i \cdot \mathbf{D}^\perp < 0$.

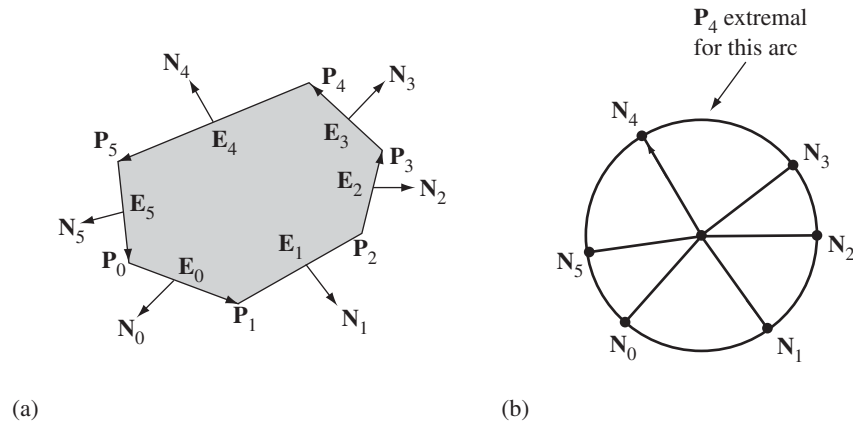


Figure 8.3 (a) A convex polygon. (b) A unit circle whose vertices correspond to normal directions of the polygon and whose arcs connecting the vertices correspond to vertices of the polygon (the *polar dual* of the polygon).

Generally there are n arcs for an n -sided polygon. We could search the arcs one at a time and test if $\mathbf{D}$ is on that arc, but then we are back to an $O(n)$ search. Instead, we can create a BSP tree for the polar dual that supports an $O(\log n)$ search. A simple illustration using the polygon of Figure 8.3 suffices. Figure 8.4 illustrates the construction of the BSP tree. Given a direction vector $\mathbf{D}$, suppose the sign tests take you down the path from $\mathbf{N}_3$ to $\mathbf{N}_5$ to $\mathbf{N}_4$ and then to $\mathbf{P}_5$. This indicates that $\mathbf{D}$ is left of $\mathbf{N}_3$, right of $\mathbf{N}_5$, and left of $\mathbf{N}_4$. This places $\mathbf{D}$ on the arc from $\mathbf{N}_4$ to $\mathbf{N}_5$, in which case $\mathbf{P}_5$ is the extremal vertex in the specified direction.

Extremal Query for a Convex Polyhedron

A convex polyhedron has vertices $\mathbf{P}_i$ for $0 \leq i < n$ and a set of edges and a set of faces with outer-pointing normals $\mathbf{N}_j$. The set of extremal points for a specified direction is either a polyhedron vertex, edge, or face. To illustrate, Figure 8.5 shows a tetrahedron and a unit sphere with vertices that correspond to the face normals of the tetrahedron, whose great circle arcs connecting the vertices correspond to the edges of the tetrahedron, and whose spherical polygons correspond to the vertices of the tetrahedron. This view of the sphere is called the *spherical dual* of the polyhedron.

The tetrahedron has vertices $\mathbf{P}_0 = (0, 0, 0)$, $\mathbf{P}_1 = (1, 0, 0)$, $\mathbf{P}_2 = (0, 1, 0)$, and $\mathbf{P}_3 = (0, 0, 1)$. The face normals are $\mathbf{N}_0 = (1, 1, 1)/\sqrt{3}$, $\mathbf{N}_1 = (-1, 0, 0)$, $\mathbf{N}_2 = (0, -1, 0)$, and $\mathbf{N}_3 = (0, 0, -1)$. The sphere is partitioned into four spherical triangles. The interior of the spherical triangle with $\langle \mathbf{N}_0, \mathbf{N}_1, \mathbf{N}_2 \rangle$ corresponds to those directions

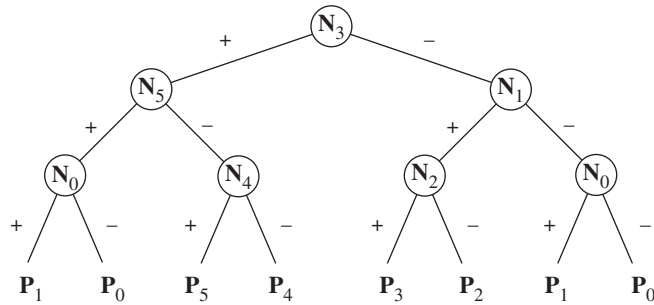


Figure 8.4 A BSP tree constructed by recursive subdivision of the unit disk. The left child of node N_j is marked with a + to indicate $N_j \cdot D^\perp \geq 0$. All normal vectors of the nodes in the left subtree are left of N_j . The right child is marked with a - to indicate $N_j \cdot D^\perp < 0$. All normal vectors of the nodes in the right subtree are right of N_j . Each node N_j represents the normal for which j is the median value of the indices represented by all nodes in the subtree rooted at N_j .

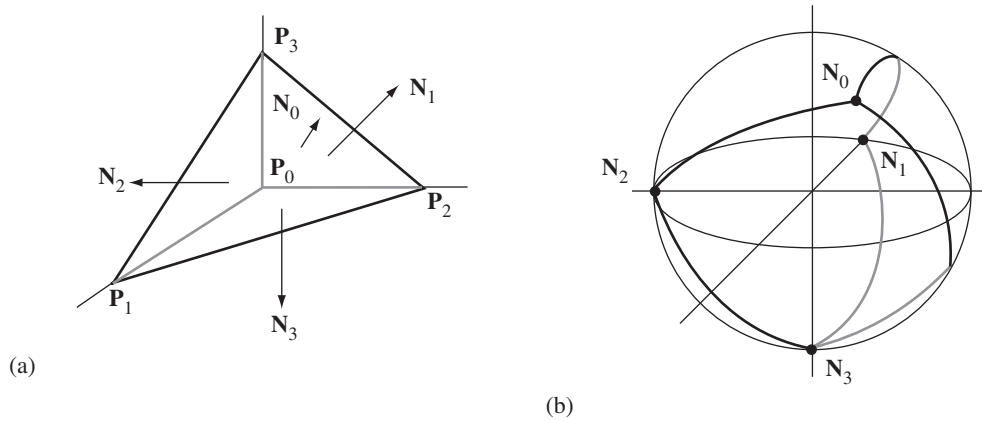


Figure 8.5 (a) A tetrahedron. (b) A unit sphere whose vertices correspond to normal directions of the tetrahedron, whose great circle arcs connecting the vertices correspond to edges of the tetrahedron, and whose spherical polygons correspond to vertices of the tetrahedron (the *spherical dual* of the tetrahedron).

for which P_3 is the unique extreme point. Observe that the three normals forming the spherical triangle are the normals for the faces that share vertex P_3 .

Generally, the normal and edge directions of a polytope lead to a partitioning of the sphere into spherical convex polygons. The interior of a single spherical convex polygon corresponds to the set of directions for which a vertex of the polytope is the unique extreme point. The number of edges of the spherical convex polygon is the number of polytope faces sharing that vertex. Just as for convex polygons in 2D, we can construct a BSP tree of the spherical polygons and use it for fast determination of extreme vertices. The method used for 2D extends to 3D with each node of the BSP tree representing a hemisphere determined by $N_i \times N_j \cdot D \geq 0$, where N_i and N_j are unit-length normal vectors for two adjacent triangles.

The vector $H_{ij} = N_i \times N_j$ is perpendicular to the plane containing the two normals. The hemispheres corresponding to this vector are $H_{ij} \cdot D \geq 0$ and $H_{ij} \cdot D < 0$. The tetrahedron of Figure 8.5 has six such vectors, listed as $\{H_{12}, H_{13}, H_{23}, H_{01}, H_{02}, H_{03}\}$. Please note that the subscripts correspond to normal vector indices, not to vertex indices. Each arc of the sphere connecting two normal vectors corresponds to an edge of the tetrahedron; let's label the arcs A_{ij} . The root node of the tree claims arc A_{12} for splitting. The condition $N_1 \times N_2 \cdot D \geq 0$ splits the sphere into two hemispheres. Figure 8.6 shows those hemispheres with viewing direction $(0, 0, -1)$.

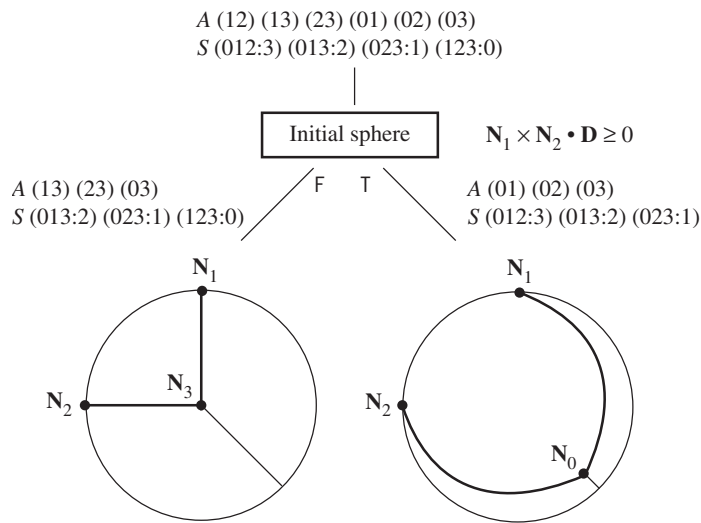


Figure 8.6 The root of the BSP tree and the two hemispheres obtained by splitting. Both children are displayed with a viewing direction $(0, 0, -1)$. The right child is the top of the sphere viewed from the outside and the left child is the bottom of the sphere viewed from the inside.

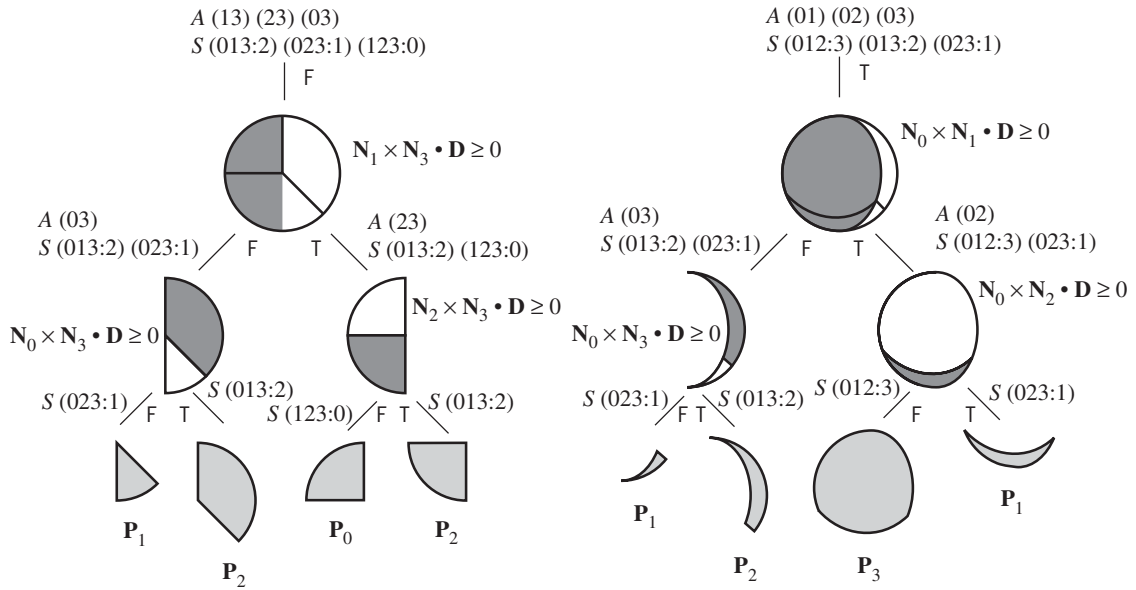


Figure 8.7 The BSP trees for the children of the root. The algebraic test is listed next to each root. The links to the children are labeled with T when the test is true or labeled with F when the test is false.

The set of arcs and the set of spherical polygons bounded by the arcs are the inputs to the BSP tree construction. These sets are shown at the top of the figure. An arc is specified by A_{ij} and connects $\mathbf{N}_i$ and $\mathbf{N}_j$. A spherical polygon is $S_{i_1, \dots, i_n; \ell}$ and has vertices $\mathbf{N}_{i_1}$ through $\mathbf{N}_{i_n}$. The vertex $\mathbf{P}_\ell$ of the original polyhedron is the extreme vertex corresponding to the spherical polygon. In our example the spherical polygons all have three vertices. Figure 8.7 shows the BSP trees for the children of the root.

During BSP tree construction, the root node claims the first arc A_{12} and uses the vector $\mathbf{H} = \mathbf{N}_1 \times \mathbf{N}_2$ for testing other vectors corresponding to arcs A_{ij} . Let $d_i = \mathbf{H} \cdot \mathbf{N}_i$ and $d_j = \mathbf{H} \cdot \mathbf{N}_j$. If $d_i \geq 0$ and $d_j \geq 0$, then the arc is completely on one side of the hemisphere implied by $\mathbf{E}$. A_{ij} is placed in a set that will be used to generate the BSP tree for the right child of the root. If $d_i \leq 0$ and $d_j \leq 0$, then the arc is completely on the other side of the hemisphere and is placed in a set that will be used to generate the BSP tree for the left child of the root. If $d_i d_j < 0$, then the arc is partially in each hemisphere and is added to both sets. This is exactly the algorithm we used in 2D.

In 3D we have some additional work in that the spherical faces must be processed by the tree to propagate to the leaf nodes the indices of the extreme vertices represented by those nodes. In fact, the processing is similar to that for arcs. Let $S_{i,j,k;\ell}$ be a face to be processed at the root node. Let $d_i = \mathbf{E} \cdot \mathbf{N}_i$, $d_j = \mathbf{E} \cdot \mathbf{N}_j$, and $d_k = \mathbf{E} \cdot \mathbf{N}_k$.

If $d_i \geq 0$ and $d_j \geq 0$ and $d_k \geq 0$, then the spherical face is completely on one side of the hemisphere implied by $\mathbf{E}$. $S_{i,j,k;\ell}$ is placed in a set that will be used to generate the BSP tree for the right child of the root. If $d_i \leq 0$, $d_j \leq 0$, and $d_k \leq 0$, then the face is completely on the other side of the hemisphere and is placed in a set that will be used to generate the BSP tree for the left child of the root. Otherwise, the arc is partially in each hemisphere and is added to both sets. In general for a spherical face with n vertices, the face is used for construction of the right child if all dot products are nonnegative, for construction of the left child if all dot products are nonpositive, or for construction of both children if some dot products are positive and some are negative.

A query for a specified direction $\mathbf{D}$ is structured the same as for convex polygons. The signs of the dot products of $\mathbf{D}$ with the $\mathbf{H}$ vectors in the BSP tree are computed, and the appropriate path is taken through the tree. A balanced tree will have depth $O(\log n)$ for a polyhedron of n vertices, so the query takes logarithmic time. However, there is a technical problem. The spherical arcs as described so far might not lead to a balanced tree. Consider a polyhedron formed by an $(n - 2)$ -sided convex polygon in the xy -plane with vertices $\mathbf{P}_i = (x_i, y_i, 0)$ for $1 \leq i \leq n - 2$ and by two vertices $\mathbf{P}_0 = (0, 0, z_0)$, with $z_0 < 0$, and $\mathbf{P}_{n-1} = (0, 0, z_{n-1})$, with $z_{n-1} > 0$. Figure 8.8 shows such a polyhedron.

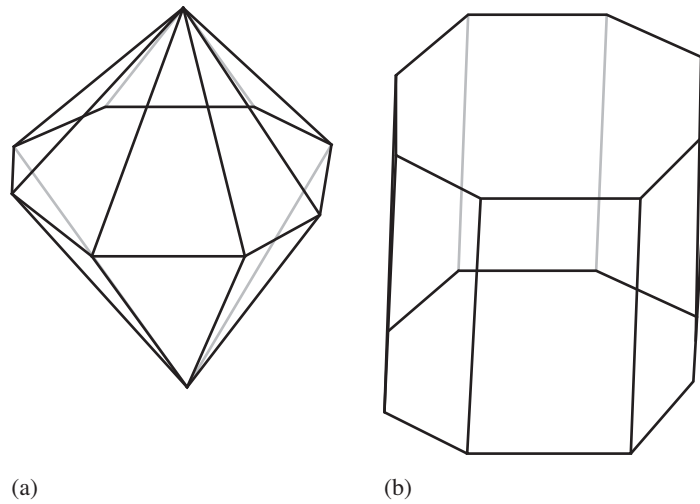


Figure 8.8 (a) A convex polyhedron for which the point-in-spherical-polygon test, using only the original arcs, is $O(n)$. (The figure shows $n = 8$, but imagine a very large n .) (b) The inscribed convex polyhedron whose edges generate the arcs of the spherical convex polygons.

The spherical dual has two spherical convex polygons, each with $n - 2$ arcs and $n - 2$ spherical convex polygons, each with 4 arcs. If $\mathbf{D}$ is contained by one of the $(n - 2)$ -sided spherical polygons, the determination of this using only the given arcs requires $n - 2$ point-on-which-side-of-arc queries. This is an $O(n)$ algorithm.

Obtaining $O(\log n)$ Queries

The pathological problem mentioned previously is avoided by appealing to an $O(\log n)$ query for point-in-convex-polygon determination. In fact, this problem uses what you may think of as the Dobkin-Kirkpatrick hierarchy restricted to two dimensions. However, it is phrased in terms of a binary search using binary separating lines. Consider the convex polygon of Figure 8.9.

If we use only the polygon edges for containment testing, we would need six tests, each showing that the query point is to the left of the edges. Instead, we use *bisectors*. The first bisector is segment $\langle \mathbf{P}_0, \mathbf{P}_3 \rangle$ and is drawn in taupe in the figure. The query point $\mathbf{Q}$ is either to the left of the bisector, where $(\mathbf{P}_3 - \mathbf{P}_0) \cdot (\mathbf{Q} - \mathbf{P}_0)^\perp \geq 0$, or to the right of the bisector, where $(\mathbf{P}_3 - \mathbf{P}_0) \cdot (\mathbf{Q} - \mathbf{P}_0)^\perp < 0$. The original polygon has six edges. The polygon to the left of the bisector has four edges, and the containment test is applied to that left polygon. The bisector edge has already been tested, so only the three remaining edges need to be tested for the containment.

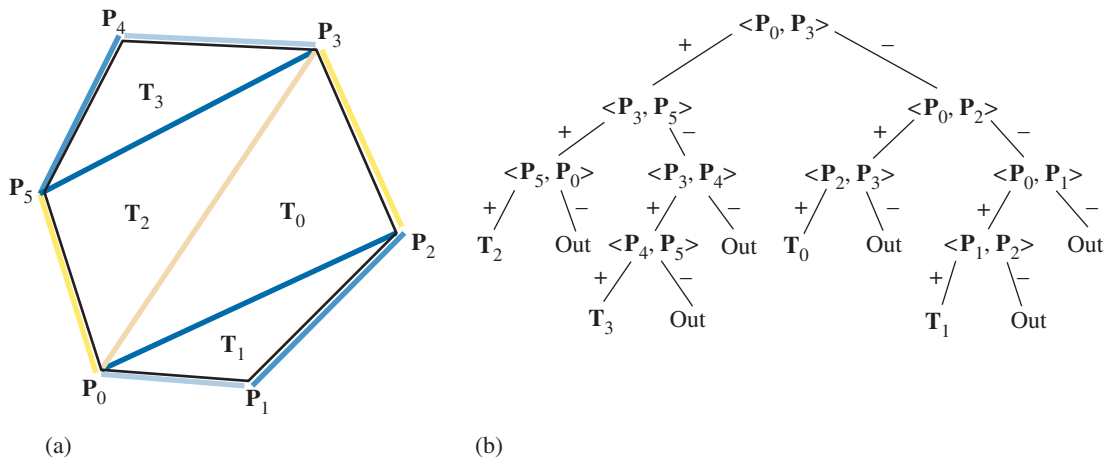


Figure 8.9 A convex polygon with bisectors used for an $O(\log n)$ point-in-polygon query. The polygon is partitioned into four triangles labeled T_0 through T_3 .

The next bisector to be used is one of the segments drawn in dark blue, either $\langle \mathbf{P}_3, \mathbf{P}_5 \rangle$ or $\langle \mathbf{P}_0, \mathbf{P}_2 \rangle$. Naturally, the choice depends on which side of the bisector $\langle \mathbf{P}_0, \mathbf{P}_3 \rangle$ the query point $\mathbf{Q}$ occurs. If $\mathbf{Q}$ is to the left of $\langle \mathbf{P}_0, \mathbf{P}_3 \rangle$ and to the left of $\langle \mathbf{P}_3, \mathbf{P}_5 \rangle$, then the only subpolygon that might contain $\mathbf{Q}$ is a triangle. The remaining edge to test is $\langle \mathbf{P}_5, \mathbf{P}_0 \rangle$. If, instead, $\mathbf{Q}$ is to the right of $\langle \mathbf{P}_3, \mathbf{P}_5 \rangle$, then $\mathbf{Q}$ is potentially in triangle T_3 . There are two remaining edges to test, so we can consider this yet another bisection step.

The binary tree in Figure 8.9(b) shows the query tree implied by the bisectors and polygon edges. A tree link marked with a + indicates “to the left of” and a link marked with a – indicates “to the right of.” The query point is tested against each edge, leading to a path from the root of the tree to a leaf node. The leaves are labeled with the region defined by the path. Four of the leaf nodes represent the triangles that make up the convex polygon. Six of the leaf nodes represent the exterior of the polygon; that is, a query point can be “outside” one of the six edges of the polygon.

The choice of the bisectors is based on selecting the medians of the ranges of indices for the subpolygon of interest. This leads to a balanced tree, so a polygon of n edges. The point-in-polygon query requires computing vector differences and dot products for a linear path of nodes through the tree. Such a path has $O(\log n)$ nodes.

The same idea may be applied to determining whether a unit-length vector $\mathbf{D}$ is contained in a spherical convex polygon on the unit sphere. In the 2D problem, the dot product whose sign determines which side of the bisector the point is on was of the form

$$d = (\mathbf{P}_i - \mathbf{P}_j) \cdot (\mathbf{Q} - \mathbf{P}_j)^\perp$$

We cared about $d \geq 0$ or $d < 0$. In the 3D problem, we use the normal vectors as the vertices and the direction vector as the query point. The dot product of interest is

$$d = \mathbf{N}_i \times \mathbf{N}_j \cdot \mathbf{D}$$

Imagine walking along the spherical arc from $\mathbf{N}_j$ to $\mathbf{N}_i$. The vector $\mathbf{N}_i \times \mathbf{N}_j$ points to your left as you walk along the arc. The spherical point $\mathbf{D}$ is to your left whenever $d \geq 0$ and is to your right whenever $d < 0$.

What this means is that the BSP tree we build for the spherical dual must use the bisectors of the spherical polygons as well as the arcs that are their boundary edges. Moreover, we do not have just one spherical polygon to test, we have n such polygons—one for each of the n vertices of the original convex polyhedron. Given the collection of all bisector arcs and boundary edge arcs, we can build the BSP tree one arc at a time. Each arc is tested at a node of the BSP tree to see on which side of the node arc it lies. If fully on the left side, we send the arc to the left subtree of the node for further classification. If fully on the right side, we send the arc to the right subtree. When the arc arrives at a leaf node, it is similarly tested for sidedness and stored as the appropriate child of the leaf node (which now becomes an interior node). If an arc straddles the great circle containing the node arc—one arc endpoint is to the left

of the node arc and one arc endpoint is to the right of the node arc—we send the arc to both subtrees of the node. That is, no actual splitting of the arc is performed. This avoids expensive arc-arc intersection finding.

The order of the arcs is important in determining the structure of the BSP tree. We do want a balanced tree. A heuristic to obtain a balanced tree is to sort the arcs. The bisector arcs occur first, the boundary edge arcs last. The first bisector arc of an n -sided spherical polygon splits the polygon into two subpolygons, each with half the number of boundary edge arcs. If we were to do a point-in-spherical-polygon query using such a bisector edge first, we will eliminate half of that spherical polygon's bisectors and half of its boundary arcs from further processing. This suggests that we order the bisector arcs based on how many other arcs they reject during a sidedness test. In the implementation, I maintain an ordered set of arcs, using a *separation* measure. The first bisector arc for an n -sided spherical polygon has a separation of $n/2$, measuring how many boundary arcs of that polygon separate the endpoints. A boundary arc itself has a separation measure of 1. The BSP tree is built using the arcs in decreasing order of separation. My numerical experiments showed that indeed the BSP trees are balanced.

An Implementation and Timing

The Foundation library files `Wm4ExtremalQuery3.h` and `Wm4ExtremalQuery4.cpp` are the base class for extremal queries for convex polyhedra. The straightforward $O(n)$ method for computing the extreme points just involves projecting the vertices onto the specified direction vector and computing the extreme projection values. This algorithm is implemented in `Wm4ExtremalQuery3PRJ.h` and `Wm4ExtremalQuery4PRJ.cpp`.

The files `Wm4ExtremalQuery3BSP.h` and `Wm4ExtremalQuery3BSP.cpp` implement the BSP tree algorithm described in this document. The identification of adjacent polyhedron normal vectors $\mathbf{N}_i$ and $\mathbf{N}_j$ requires building a vertex-edge-face data structure. The class implemented in `Wm4BasicMesh.h` and `Wm4BasicMesh.cpp` suffices, but the edges adjacent to a vertex are not required for the extremal queries. You could modify `BasicMesh` to eliminate this adjacency information.

Table 8.1 shows the results of the experiment to compare the BSP tree approach to a simple project-all-vertices approach. The column with header n is the number of vertices of the convex polyhedron. Each polyhedron was used in 10^7 extremal queries. The execution times are listed in the second and third columns, and are in seconds. The target machine was an AMD Athlon XP 2800+ (2.08GHz). The next-to-last column is the BSP time divided by $\log n$. This ratio is expected to be a constant for large n ; that is, we expect the query to be $O(\log n)$. The last column is the project-all-vertices time divided by n , since we expect this algorithm to be $O(n)$.

Of interest is the break-even n . It is somewhere between 32 and 64. If the convex polyhedra in your applications have a small number of vertices, the project-all-vertices approach is clearly the choice. For a larger number of vertices, the BSP approach wins.

Table 8.1 A comparison of times between projecting all vertices to compute a projection interval and using a BSP tree for the query.

n	BSP Time t_b	Project-All Time t_p	$t_b / \log n$	t_p / n
4	2.141	0.812	1.0705	0.2030
8	3.922	1.547	1.3073	0.1933
16	5.422	2.563	1.3555	0.1601
32	5.937	4.328	1.1874	0.1352
64	6.922	7.765	1.1536	0.1213
128	7.922	14.391	1.1317	0.1124
256	9.281	27.359	1.1601	0.1068
512	10.250	53.859	1.1388	0.1051
1024	11.532	104.125	1.1532	0.1016
2048	12.797	210.765	1.1633	0.1029

A sample application to illustrate the queries is in the `SampleMiscellaneous` folder, project `ExtremalQuery`. A convex polyhedron is displayed using an orthogonal camera. You may rotate it with the mouse. The extreme vertices in the x -direction are drawn as small spheres. The orthogonal camera is used to make it clear that the points are extreme.

8.1.2 STATIONARY OBJECTS

The method of separating axes for stationary objects determines whether or not two objects intersect, a test-intersection query. We will analyze the method for convex polygons in 2D to motivate the ideas, then extend the method to convex polyhedra in 3D. The previous discussion showed us how to compute the projection intervals. Now we need to know which axes to project onto.

Convex Polygons

The following notation is used throughout this section. Let C_j for $j = 0, 1$ be the convex polygons with vertices $\mathbf{P}_i^{(j)}$ for $0 \leq i < N_j$ that are counterclockwise ordered. The edges of the polygons have direction vectors $\mathbf{E}_i^{(j)} = \mathbf{P}_{i+1}^{(j)} - \mathbf{P}_i^{(j)}$ for $0 \leq i < N_j$ and where modular indexing is used to handle wraparound (index N is the same as index 0; index -1 is the same as index $N - 1$). Outward normal vectors to the edges are $\mathbf{N}_i^{(j)}$. No assumption is made about the length of the normal vectors; an

implementation may choose the length as needed. Regardless of length, the condition of outward pointing means

$$\left(\mathbf{N}_i^{(j)}\right)^\perp \cdot \mathbf{E}_i^{(j)} > 0$$

where $(x, y)^\perp = (-y, x)$. All the pseudocode relating to convex polygons will use the class shown:

```
class ConvexPolygon
{
public:
    // N, number of vertices
    int GetN();

    // V[i], counterclockwise ordered
    Point GetVertex (int i);

    // E[i] = V[i + 1] - V[i]
    Vector GetEdge (int i);

    // N[i], N[i].x * E[i].y - N[i].y * E[i].x > 0
    Vector GetNormal (int i);
};
```

All functions are assumed to handle the wraparound. For example, if the input value is N , the number of vertices, then `GetVertex` returns $\mathbf{P}_0$ and `GetEdge` returns $\mathbf{P}_1 - \mathbf{P}_0$. If the input value is -1 , then `GetVertex` returns $\mathbf{P}_{N-1}$ and `GetEdge` returns $\mathbf{P}_0 - \mathbf{P}_{N-1}$. Only the relevant interface is supplied for clarity of presentation. The implementation details will vary with the needs of an application.

For a pair of convex polygons, only a finite set S of direction vectors needs to be considered for separation tests. That set contains only the normal vectors to the edges of the polygons. Figure 8.10 (a) shows two nonintersecting polygons that are separated along a direction determined by the normal to an edge of one polygon. Figure 8.10 (b) shows two polygons that intersect; there are no separating directions.

The intuition for why only edge normals must be tested is based on having two convex polygons just touching with no interpenetration. Figure 8.11 shows the three possible configurations: edge-edge contact, vertex-edge contact, and vertex-vertex contact. The lines between the polygons are perpendicular to the separation lines that would occur for one object translated away from the other by an infinitesimal distance. The vertex-vertex edge case has a low probability of occurrence. The collision system should report this as a vertex-face collision to be consistent with our classification of contact points (vertex-face or edge-edge with appropriately assigned normal vectors).

A naive implementation of the method of separating axes selects a potential separating direction $\mathbf{D}$, computes the intervals of projection by projecting the vertices of

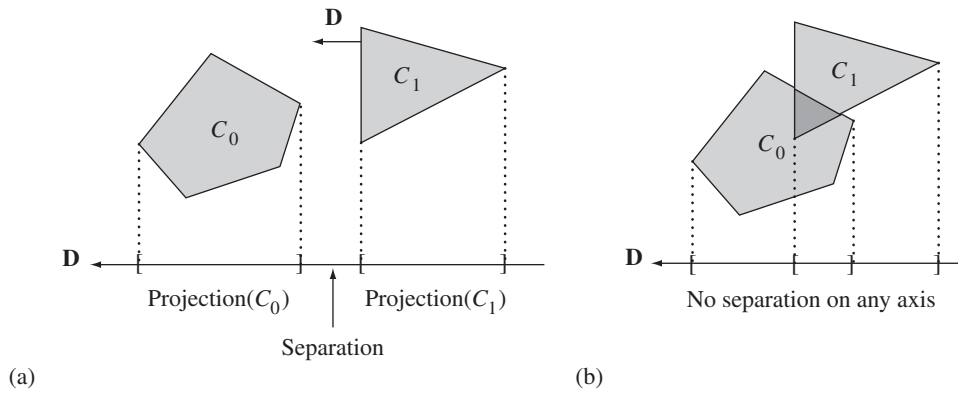


Figure 8.10 (a) Nonintersecting convex polygons. (b) Intersecting convex polygons.

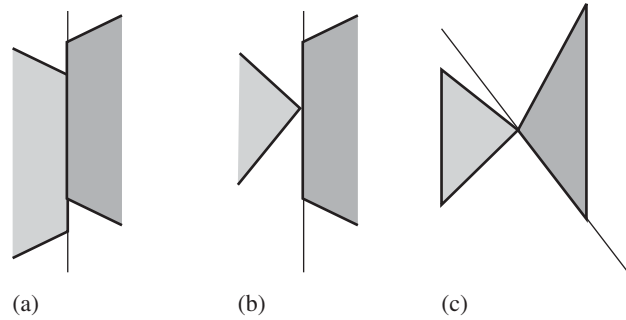


Figure 8.11 (a) Edge-edge contact. (b) Vertex-edge contact. (c) Vertex-vertex contact.

both polygons onto that line, then tests if the intervals are separated. This requires computing N_i projections for polygon C_i and keeping track of the minimum and maximum projection values for each polygon. In the worst case that the polygons intersect, N_0 directions are tested from C_0 , each requiring $N_0 + N_1$ projections, and N_1 directions are tested from C_1 , each requiring $N_0 + N_1$ projections. The total number of projections is $(N_0 + N_1)^2$.

A smarter algorithm avoids projecting all the vertices for the polygons by only testing for separation using the maximum of the interval for the first polygon and the minimum of the interval for the second polygon. If $\mathbf{D}$ is an outward-pointing normal

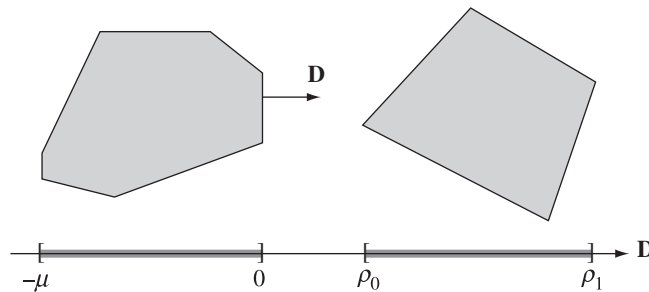


Figure 8.12 Two polygons separated by an edge-normal direction of the first polygon.

for the edge $\mathbf{P}_{i+1}^{(0)} - \mathbf{P}_i^{(0)}$ of C_0 , then the projection of the C_0 onto the separating line $\mathbf{P}_i^{(0)} + t\mathbf{D}$ is $[-\mu, 0]$, where $\mu > 0$. If the projection of C_1 onto this line is $[\rho_0, \rho_1]$, then the reduced separation test is $\rho_0 > 0$. Figure 8.12 illustrates two separated polygons using this scheme.

The value μ is irrelevant since we only need to compare ρ_0 to 0. Consequently, there is no need to project the vertices of C_0 to calculate μ . Moreover, the vertices of C_1 are projected one at a time until either the projected value is negative, in which case $\mathbf{D}$ is no longer considered for separation, or until all projected values are positive, in which case $\mathbf{D}$ is a separating direction.

The pseudocode for the algorithm is

```
bool TestIntersection (ConvexPolygon C0, ConvexPolygon C1)
{
    // Test edges of C0 for separation. Because of the
    // counterclockwise ordering, the projection interval for
    // C0 is [m,0] where m ≤ 0. Try only to determine if C1
    // is on the 'positive' side of the line.
    for (i0 = C0.GetN() - 1, i1 = 0; i1 < C0.GetN(); i0 = i1++)
    {
        P = C0.GetVertex(i1);
        D = C0.GetNormal(i0);
        if (WhichSide(C1,P,D) > 0)
        {
            // C1 is entirely on the 'positive' side of line P+t*D
            return false;
        }
    }
}
```

```

    // Test edges of C1 for separation. Because of the
    // counterclockwise ordering, the projection interval for
    // C1 is [m,0] where m <= 0. Try only to determine if C0
    // is on the 'positive' side of the line.
    for (i0 = C1.GetN() - 1, i1 = 0; i1 < C1.GetN(); i0 = i1++)
    {
        P = C1.GetVertex(i1);
        D = C1.GetNormal(i0);
        if (WhichSide(C0,P,D) > 0)
        {
            // C0 is entirely on the 'positive' side of line P+t*D
            return false;
        }
    }

    return true;
}

int WhichSide (ConvexPolygon C, Point P, Vector D)
{
    // C vertices are projected onto line P+t*D. Return value
    // is +1 if all t > 0, -1 if all t < 0, or 0 if the line
    // splits the polygon.

    posCount = 0;
    negCount = 0;
    zeroCount = 0;
    for (i = 0; i < C.GetN(); i++)
    {
        t = Dot(D,C.GetVertex(i) - P);
        if (t > 0)
        {
            posCount++;
        }
        else if (t < 0)
        {
            negCount++;
        }
        else
        {
            zeroCount++;
        }
    }

    if ((posCount > 0 and negCount > 0) or zeroCount > 0)

```

```

        {
            return 0;
        }
    }
    return posCount > 1 ? -1;
}

```

In the worst case, the polygons do intersect. We have processed N_0 edge normals of C_0 , each requiring N_1 projections for C_1 , and N_1 edge normals of C_1 , each requiring N_0 projections for C_0 . The total number of projections is $2N_0N_1$, still a quadratic quantity but considerably smaller than $(N_0 + N_1)^2$.

We can do even better in an asymptotic sense as the number of vertices becomes large, using the extremal query for convex polygons mentioned in Section 8.1.1. For a polygon of N_0 vertices, the bisection is of order $O(\log N_0)$, so the total algorithm is $O(\max\{N_0 \log N_1, N_1 \log N_0\})$. However, in practice the values of N_0 and N_1 are sufficiently small that the asymptotic performance is not relevant.

Convex Polyhedra

The following notation is used throughout this section. Let C_j for $j = 0, 1$ be the convex polyhedra with vertices $\mathbf{P}_i^{(j)}$ for $0 \leq i < N_j$, edges with directions $\mathbf{E}_i^{(j)}$ for $0 \leq i < M_j$, and faces that are planar convex polygons whose vertices are ordered counterclockwise as you view the face from outside the polyhedron. The outward normal vectors for the faces are $\mathbf{N}_i^{(j)}$ for $0 \leq i < L_j$. All the pseudocode relating to convex polyhedra will use the class shown next.

```

class ConvexPolyhedron
{
public:
    int GetVCount (); // number of vertices
    int GetECount (); // number of edges
    int GetFCount (); // number of faces
    Point GetVertex (int i);
    Vector GetEdge (int i);
    Vector GetNormal (int i);
};

```

Only the relevant interface is supplied for clarity of presentation. The implementation details will vary with the needs of an application.

The ideas of separation of convex polygons extend to convex polyhedra. For a pair of convex polyhedra, only a finite set of direction vectors needs to be considered for separating tests. The intuition is similar to that of convex polygons. If the two polyhedra are just touching with no interpenetration, the contact is one of face-face,

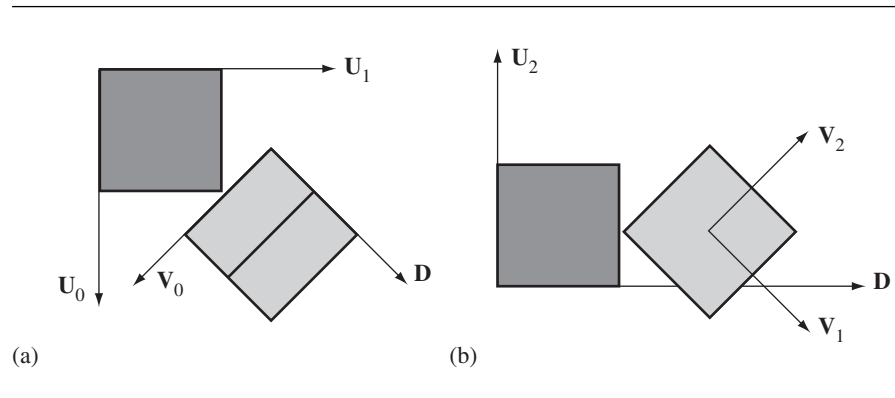


Figure 8.13 Two views of two cubes that are not separated by any face normal but are separated by a cross product of two edges, one from each cube.

face-edge, face-vertex, edge-edge, edge-vertex, or vertex-vertex. The set of potential separating directions that capture these types of contact includes the normal vectors to the faces of the polyhedra and vectors generated by a cross product of two edges, one from each polyhedron. The necessity of testing more than just the face normals is shown by Figure 8.13.

The first cube in the figure (dark gray) has unit-length face normals $U_0 = (1, 0, 0)$, $U_1 = (0, 1, 0)$, and $U_2 = (0, 0, 1)$. The second cube (light gray) has unit-length face normals $V_0 = (1, -1, 0)/\sqrt{2}$, $V_1 = (1, 1, -\sqrt{2})/2$, and $V_2 = (1, 1, \sqrt{2})/2$. The other vector shown is $D = (1, 1, 0)/\sqrt{2}$. Figure 8.13 (a) shows a view of the two cubes when looking in the direction $-U_2$. Figure 8.13 (b) shows a view when looking along the direction $U_2 \times D$. In view (a), neither U_0 , U_1 , nor V_0 are separating directions. In view (b), neither U_2 , V_1 , nor V_2 are separating directions. No face axis separates the two cubes, yet they are not intersecting. A separating direction is $D = U_2 \times V_0$, a cross product of edges from the cubes.

The pseudocode for using the method of separating axes to test for intersection of two polyhedra, similar to the naive implementation in 2D, is

```
bool TestIntersection (ConvexPolyhedron C0,
    ConvexPolyhedron C1)
{
    // Test faces of C0 for separation.
    for (i = 0; i < C0.GetFCount(); i++)
    {
        D = C0.GetNormal(i);
        ComputeInterval(C0,D,min0,max0);
        ComputeInterval(C1,D,min1,max1);
```

```

        if (max1 < min0 || max0 < min1)
        {
            return false;
        }
    }

    // Test faces of C1 for separation.
    for (j = 0; j < C1.GetFCount(); j++)
    {
        D = C1.GetNormal(j);
        ComputeInterval(C0,D,min0,max0);
        ComputeInterval(C1,D,min1,max1);
        if (max1 < min0 || max0 < min1)
        {
            return false;
        }
    }

    // Test cross products of pairs of edges.
    for (i = 0; i < C0.GetECount(); i++)
    {
        for (j = 0; j < C1.GetECount(); j++)
        {
            D = Cross(C0.GetEdge(i),C1.Edge(j));
            ComputeInterval(C0,D,min0,max0);
            ComputeInterval(C1,D,min1,max1);
            if (max1 < min0 || max0 < min1)
            {
                return false;
            }
        }
    }

    return true;
}

void ComputeInterval (ConvexPolyhedron C, Vector D,
double& min, double& max)
{
    min = Dot(D,C.GetVertex(0));
    max = min;
    for (i = 1; i < C.GetVCount(); i++)
    {
        value = Dot(D,C.GetVertex(i));

```

```

        if (value < min)
        {
            min = value;
        }
        else
        {
            max = value;
        }
    }
}

```

The function `ComputeInterval` is $O(N)$ for a polyhedron of N vertices. A quick glance at the code shows that you need $L_0(N_0 + N_1) + L_1(N_0 + N_1) + M_0M_1(N_0 + N_1) = (L_0 + L_1 + M_0M_1)(N_0 + N_1)$ units of time to execute the test (cubic order). Since all pairs of edges are tested in the worst case, the time is at least quadratic. This function can also be implemented to use the asymptotically faster extremal query using BSP trees; see Section 8.1.1. As mentioned in that section, though, the performance is better only for a sufficiently large number of polyhedron vertices. You can always add a conditional statement to `ComputeInterval` that switches between projecting all vertices and projecting using the extremal query based on the number of vertices.

8.1.3 OBJECTS MOVING WITH CONSTANT LINEAR VELOCITY

The method of separating axes is used to test for intersection of two stationary convex polyhedra. The set of potential separating axes consists of the face normals for the polyhedra and cross products of edges, each product using an edge from each of the participating polyhedra. As described earlier, a collision detection system can be implemented using a bisection technique. At the first time step, no polyhedra are intersecting. For the next time step, the physical simulation decides how each polyhedron should move based on constraints (using a differential equation solver for the equations of motion). All the polyhedra are moved to their desired locations. Each pair of convex polyhedra are tested for intersection using the method of separating axes. If any pair reports interpenetration, then the system is restarted at the first time step, but with a time increment that is half of what was tried the first time.

The strategy is not bad when only a small number of polyhedra are in the system and when the frequency of contact (or close contact) is small. However, for large numbers of polyhedra in close proximity, a lot of time can be spent in restarting the system. A quick hack to reduce the time is to restart the system only for those pairs that report an interpenetration. When the bisection is completed, all the objects have been moved to a new state with no interpenetrations, but the time step is (potentially) different for the objects, an implied change in the speeds of some of the objects. For a system that runs on the order of 30 frames per second or larger, this is not usually a noticeable problem.

An alternative to the bisection approach is to attempt to predict the time of collision between two polyhedra. For a small change in time, an assumption we can make is that the polyhedra are moving with constant linear velocity and zero angular velocity. Whether or not the assumption is reasonable will depend on your application. The method of separating axes can be extended to handle polyhedra moving with constant linear velocity and to report the first time of contact between a pair. The algorithm is attributed to Ron Levine in a post to the SourceForge game developer algorithms mailing list [Lev00]. As we did for stationary objects, let us first look at the problem for convex polygons to illustrate the ideas for convex polyhedra.

Separation of Convex Polygons

If C_0 and C_1 are convex polygons with linear velocities $\mathbf{V}_0$ and $\mathbf{V}_1$, it can be determined via projections if the polygons will intersect for some time $T \geq 0$. If they do intersect, the first time of contact can be computed. It is enough to work with a stationary polygon C_0 and a moving polygon C_1 with velocity $\mathbf{V}$ since you can always use $\mathbf{V} = \mathbf{V}_1 - \mathbf{V}_0$ to perform the calculations as if C_0 were not moving.

If C_0 and C_1 are initially intersecting, then the first time of contact is $T = 0$. Otherwise, the convex polygons are initially disjoint. The projection of C_1 onto a line with direction $\mathbf{D}$ not perpendicular to $\mathbf{V}$ is itself moving. The speed of the projection along the line is $\sigma = (\mathbf{V} \cdot \mathbf{D})/|\mathbf{D}|^2$. If the projection interval of C_1 moves away from the projection interval of C_0 , then the two polygons will never intersect. The cases when intersection might happen are those when the projection intervals for C_1 move toward those of C_0 .

The intuition for how to predict an intersection is much like that for selecting the potential separating directions in the first place. If the two convex polygons intersect at a first time $T_{\text{first}} > 0$, then their projections are not separated along any line at that time. An instant before first contact, the polygons are separated. Consequently, there must be at least one separating direction for the polygons at time $T_{\text{first}} - \varepsilon$ for small $\varepsilon > 0$. Similarly, if the two convex polygons intersect at a last time $T_{\text{last}} > 0$, then their projections are also not separated at that time along any line, but an instant after last contact, the polygons are separated. And again, there must be at least one separating direction for the polygons at time $T_{\text{last}} + \varepsilon$ for small $\varepsilon > 0$. Both T_{first} and T_{last} can be tracked as each potential separating axis is processed. After all directions are processed, if $T_{\text{first}} \leq T_{\text{last}}$, then the two polygons do intersect with first contact time T_{first} . It is also possible that $T_{\text{first}} > T_{\text{last}}$, in which case the two polygons cannot intersect.

Pseudocode for testing for intersection of two moving convex polygons is given next. The time interval over which the event is of interest is $[0, T_{\text{max}}]$. If knowing an intersection at *any* future time is desired, then set $T_{\text{max}} = \infty$. Otherwise, T_{max} is finite. The function is implemented to indicate there is no intersection on $[0, T_{\text{max}}]$, even though there might be an intersection at some time $T > T_{\text{max}}$.


```

bool TestIntersection (ConvexPolygon C0, Vector V0,
    ConvexPolygon C1, Vector V1, double tmax, double& tfirst,
    double& tlast)
{
    // Process as if C0 is stationary, C1 is moving.
    V = V1 - V0;
    tfirst = 0;
    tlast = INFINITY;

    // Test edges of C0 for separation.
    for (i0 = C0.GetN() - 1, i1 = 0; i1 < C0.GetN(); i0 = i1++)
    {
        D = C0.GetNormal(i0);
        ComputeInterval(C0,D,min0,max0);
        ComputeInterval(C1,D,min1,max1);
        speed = Dot(D,V);
        if (NoIntersect(tmax,speed,min0,max0,min1,max1,
            tfirst,tlast))
        {
            return false;
        }
    }

    // Test edges of C1 for separation.
    for (i0 = C1.N - 1, i1 = 0; i1 < C1.N; i0 = i1++)
    {
        D = C1.GetNormal(i0);
        ComputeInterval(C0,D,min0,max0);
        ComputeInterval(C1,D,min1,max1);
        speed = Dot(D,V);
        if (NoIntersect(tmax,speed,min0,max0,min1,max1,
            tfirst,tlast))
        {
            return false;
        }
    }
    return true;
}

bool NoIntersect (double tmax, double speed, double min0,
    double max0, double min1, double max1, double& tfirst,
    double& tlast)
{
    if (max1 < min0)

```

```

{
    // Interval(C1) is initially on 'left' of interval(C0).

    if (speed <= 0) // intervals moving apart
    {
        return true;
    }

    t = (min0 - max1)/speed;
    if (t > tfirst)
    {
        tfirst = t;
    }
    if (tfirst > tmax)
    {
        return true;
    }

    t = (max0 - min1)/speed;
    if (t < tlast)
    {
        tlast = t;
    }
    if (tfirst > tlast)
    {
        return true;
    }
}
else if (max0 < min1)
{
    // Interval(C1) is initially on 'right' of interval(C0).

    if (speed >= 0) // intervals moving apart
    {
        return true;
    }

    t = (max0 - min1)/speed;
    if (t > tfirst)
    {
        tfirst = t;
    }
    if (tfirst > tmax)
    {

```

```

        return true;
    }

    t = (min0 - max1)/speed;
    if (t < tlast)
    {
        tlast = t;
    }
    if (tfirst > tlast)
    {
        return true;
    }
}
else
{
    // Interval(C0) and interval(C1) overlap.

    if (speed > 0)
    {
        t = (max0 - min1)/speed;
        if (t < tlast)
        {
            tlast = t;
        }
        if (tfirst > tlast)
        {
            return true;
        }
    }
    else if (speed < 0)
    {
        t = (min0 - max1)/speed;
        if (t < tlast)
        {
            tlast = t;
        }
        if (tfirst > tlast)
        {
            return true;
        }
    }
}
return false;
}

```

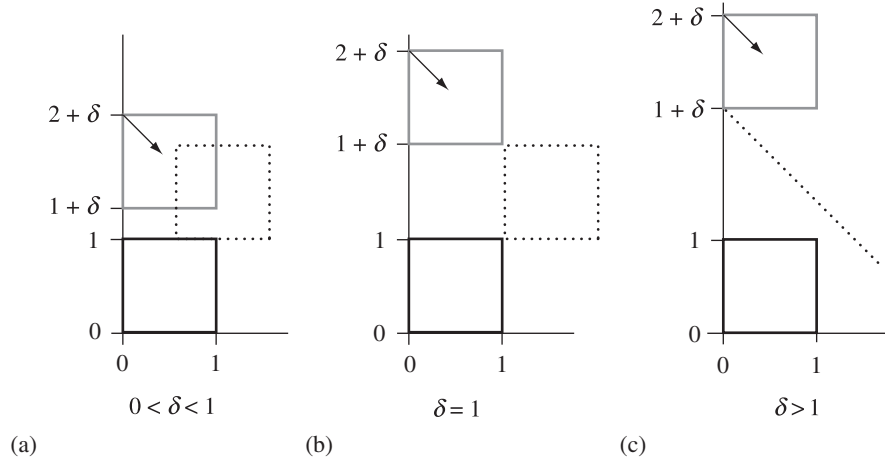


Figure 8.14 (a) Edge-edge intersection predicted. (b) Vertex-vertex intersection predicted. (c) No intersection predicted.

The function `ComputeInterval(C,D,min,max)` computes the projection interval $[\min, \max]$ of a convex polygon C onto the line of direction D using the fast extremal queries described earlier that use an approach based on BSP trees. The pseudocode as written projects the convex polygons onto the line tD . In an implementation, you most likely will want to avoid floating-point problems in the projection values when the vertices have large components. An additional parameter to `ComputeInterval` should be a point approximately near one (or both) polygons, something as simple as choosing a vertex P of a polygon. The projection is onto $P + tD$ instead.

The following example illustrates these ideas. The first box is the unit cube $0 \leq x \leq 1$ and $0 \leq y \leq 1$ and is stationary. The second box is initially $0 \leq x \leq 1$ and $1 + \delta \leq y \leq 2 + \delta$ for some $\delta > 0$. Let its velocity be $(1, -1)$. Whether or not the second box intersects the first box depends on the value of δ . The only potential separating axes are $(1, 0)$ and $(0, 1)$. Figure 8.14 shows the initial configuration for three values of δ , one where there is an edge-edge intersection, one where there is a vertex-vertex intersection, and one where there is no intersection.

The black box is stationary. The gray box is moving. The black vector indicates the direction of motion. The dotted boxes indicate where the moving box first touches the stationary box. In Figure 8.14 (c), the dotted line indicates that the moving box will miss the stationary box. For $D = (1, 0)$, the pseudocode produces $\min_0 = 0, \max_0 = 1, \min_1 = 0, \max_1 = 1$, and $\text{speed} = 1$. The projected intervals are initially overlapping. Since the speed is positive, $T = (\max_0 - \min_1) / \text{speed} = 1 < T_{\text{Last}} = \text{INFINITY}$ and T_{Last} is updated to 1. For $D = (0, 1)$, the pseudocode produces $\min_0 = 0, \max_0 = 1$,

$\text{min1} = 1 + \delta$, $\text{max1} = 2 + \delta$, and $\text{speed} = -1$. The moving projected interval is initially on the right of the stationary projected interval. Since the speed is negative, $T = (\text{max0} - \text{min1})/\text{speed} = \delta > T_{\text{first}} = 0$ and T_{first} is updated to δ . The next block of code sets $T = (\text{min0} - \text{max1})/\text{speed} = 2 + \delta$. The value T_{last} is not updated since $2 + \delta < 1$ cannot happen for $\delta > 0$. On exit from the loop over potential separating directions, $T_{\text{first}} = \delta$ and $T_{\text{last}} = 1$. The objects intersect if and only if $T_{\text{first}} \leq T_{\text{last}}$, or $\delta \leq 1$. This condition is consistent with the images in Figure 8.14. Figure 8.14 (a) has $\delta < 1$ and Figure 8.14 (b) has $\delta = 1$, intersections occurring in both cases. Figure 8.14 (c) has $\delta > 1$ and no intersection occurs.

Contact Set for Convex Polygons

Although we are interested in nonpenetration intersections for moving objects, I mention the stationary case just for completeness. The find-intersection query for two stationary convex polygons is a special example of Boolean operations on polygons. If the polygons have N_0 and N_1 vertices, there is an intersection algorithm of order $O(N_0 + N_1)$ for computing the intersection [O'R98]. A less efficient algorithm is to clip the edges of each polygon against the other polygon. The order of this algorithm is $O(NM)$. Of course, the asymptotic analysis applies to large N and M , so the latter algorithm is potentially a good choice for triangles and rectangles.

Given two moving convex objects C_0 and C_1 , initially not intersecting and with velocities $\mathbf{V}_0$ and $\mathbf{V}_1$, we showed earlier how to compute the first time of contact T , if it exists. Assuming it does, the sets $C_0 + T\mathbf{V}_0 = \{\mathbf{X} + T\mathbf{V}_0 : \mathbf{X} \in C_0\}$ and $C_1 + T\mathbf{V}_1 = \{\mathbf{X} + T\mathbf{V}_1 : \mathbf{X} \in C_1\}$ are just touching with no interpenetration. See Figure 8.11 for the various configurations.

The `TestIntersection` function can be modified to keep track of which vertices or edges are projected to the endpoints of the projection interval. At the first time of contact, this information is used to determine how the two objects are oriented with respect to each other. If the contact is vertex-edge or vertex-vertex, then the contact set is a single point, a vertex. If the contact is edge-edge, the contact set is a line segment that contains at least one vertex. Each endpoint of the projection interval is generated by either a vertex (unique extreme) or an edge (nonunique extreme). A class to store all the relevant projection information is

```
class ProjInfo
{
public:
    double min, max; // projection interval [min,max]
    int index[2];
    bool isUnique[2];
};
```

The zero-indexed entries of the array correspond to the minimum of the interval. If the minimum is obtained from the unique extreme vertex $\mathbf{V}_i$, then `index[0]` stores i

and `isUnique[0]` is true. If the minimum is obtained from an edge E_j , then `index[0]` stores j and `isUnique[0]` stores false. The same conventions apply for the one-indexed entries corresponding to the maximum of the interval.

To support calculation of the contact set and the new configuration structure, we need to modify the extremal query, call it `GetExtremeIndex`. For stationary objects, we just need to know the index of the vertex that projects to the extreme interval value. For moving objects, we also need it to tell us whether exactly one vertex projects to the extreme or an entire edge projects to the extreme. The signature for the query is

```
int GetExtremeIndex (ConvexPolyhedron C, Vector D, bool& isUnique);
```

The return value is the index for an extreme vertex. The return value of `isUnique` is true if a single vertex projects to the extreme, but false when an entire edge projects to the extreme.

Of course, in an implementation using floating-point numbers, the test on the dot product d would use some application-specified value $\varepsilon > 0$ and replace $d > 0$ by $d > \varepsilon$ and $d < 0$ by $d < -\varepsilon$. Function `ComputeInterval` must be modified to provide more information than just the projection interval.

```
void ComputeInterval (ConvexPolyhedron C, Vector D,
    ProjInfo& info)
{
    info.index[0] = GetExtremeIndex(C,-D,info.isUnique[0]);
    info.min = Dot(D,C.GetVertex(info.index[0]));
    info.index[1] = GetExtremeIndex(C,+D,info.isUnique[1]);
    info.max = Dot(D,C.GetVertex(info.index[1]));
}
```

The `NoIntersect` function accepted as input the projection intervals for the two polygons. Now those intervals are stored in the `ProjInfo` objects, so `NoIntersect` must be modified to reflect this. In the event that there will be an intersection between the moving polygons, it is necessary that the projection information be saved for later use in determining the contact set. As a result, `NoIntersect` must keep track of the `ProjInfo` objects corresponding to the current first time of contact. Finally, the contact set calculation will require knowledge of the order of the projection intervals. `NoIntersect` will set a flag `side` with value $+1$ if the intervals intersect at the maximum of the C_0 interval and the minimum of the C_1 interval, or with value -1 if the intervals intersect at the minimum of the C_0 interval and the maximum of the C_1 interval. The modified pseudocode is

```
bool NoIntersect (double tmax, double speed, ProjInfo info0,
    ProjInfo info1, ProjInfo& curr0, ProjInfo& curr1,
    int& side, double& tfirst, double& tlast)
{
    if (info1.max < info0.min)
```

```

{
    if (speed <= 0)
    {
        return true;
    }

    t = (info0.min - info1.max)/speed;
    if (t > tfirst)
    {
        tfirst = t;
        side = -1;
        curr0 = info0;
        curr1 = info1;
    }
    if (tfirst > tmax)
    {
        return true;
    }

    t = (info0.max - info1.min)/speed;
    if (t < tlast)
    {
        tlast = t;
    }
    if (tfirst > tlast)
    {
        return true;
    }
}
else if (info0.max < info1.min)
{
    if (speed >= 0)
    {
        return true;
    }
    t = (info0.max - info1.min)/speed;
    if (t > tfirst)
    {
        tfirst = t;
        side = +1;
        curr0 = info0;
        curr1 = info1;
    }
    if (tfirst > tmax)

```

```

    {
        return true;
    }

    t = (info0.min - info1.max)/speed;
    if (t < tlast)
    {
        tlast = t;
    }
    if (tfirst > tlast)
    {
        return true;
    }
}
else
{
    if (speed > 0)
    {
        t = (info0.max - info1.min)/speed;
        if (t < tlast)
        {
            tlast = t;
        }
        if (tfirst > tlast)
        {
            return true;
        }
    }
    else if (speed < 0)
    {
        t = (info0.min - info1.max)/speed;
        if (t < tlast)
        {
            tlast = t;
        }
        if (tfirst > tlast)
        {
            return true;
        }
    }
}
return false;
}

```


With the indicated modifications, `TestIntersection` has the equivalent formulation

```
bool TestIntersection (ConvexPolygon C0, Vector V0,
    ConvexPolygon C1, Vector V1, double tmax, double& tfirst,
    double& tlast)
{
    ProjInfo info0, info1, curr0, curr1;
    // Process as if C0 were stationary and C1 were moving.
    V = V1 - V0;
    tfirst = 0;
    tlast = INFINITY;

    // Process edges of C0.
    for (i0 = C0.GetN() - 1, i1 = 0; i1 < C0.GetN(); i0 = i1++)
    {
        D = C0.GetNormal(i0);
        ComputeInterval(C0,D,info0);
        ComputeInterval(C1,D,info1);
        speed = Dot(D,v);
        if (NoIntersect(tmax,speed,info0,info1,curr0,curr1,side,
            tfirst,tlast))
        {
            return false;
        }
    }

    // Process edges of C1.
    for (i0 = C1.GetN() - 1, i1 = 0; i1 < C1.GetN(); i0 = i1++)
    {
        D = C1.GetNormal(i0);
        ComputeInterval(C0,D,info0);
        ComputeInterval(C1,D,info1);
        speed = Dot(D,v);
        if (NoIntersect(tmax,speed,info0,info1,curr0,curr1,side,
            tfirst,tlast))
        {
            return false;
        }
    }

    return true;
}
```

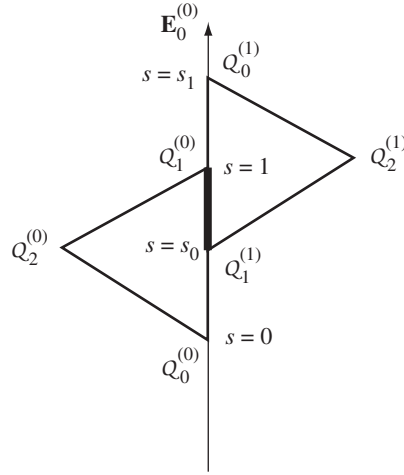


Figure 8.15 Edge-edge contact for two moving triangles.

The `FindIntersection` pseudocode has exactly the same implementation as `TestIntersection`, but with one additional block of code that is reached after the two loops if there will be an intersection. When the polygons will intersect at time T , they are effectively moved with their respective velocities and the contact set is calculated. Let $\mathbf{Q}_i^{(j)} = \mathbf{P}_i^{(j)} + T\mathbf{V}^{(j)}$ represent the polygon vertices after motion. In the case of edge-edge contact, for the sake of argument, suppose that the contact edges are $\mathbf{E}_0^{(0)}$ and $\mathbf{E}_0^{(1)}$. Figure 8.15 illustrates the configurations for two triangles.

Because of the counterclockwise ordering of the polygons, observe that the two edge directions are parallel, but in opposite directions. The edge of the first polygon is parameterized as $\mathbf{Q}_0^{(0)} + s\mathbf{E}_0^{(0)}$ for $s \in [0, 1]$. The edge of the second polygon has the same parametric form, but with $s \in [s_0, s_1]$, where

$$s_0 = \frac{\mathbf{E}_0^{(0)} \cdot (\mathbf{Q}_1^{(1)} - \mathbf{Q}_0^{(0)})}{|\mathbf{E}_0|^2} \quad \text{and} \quad s_1 = \frac{\mathbf{E}_0^{(0)} \cdot (\mathbf{Q}_0^{(1)} - \mathbf{Q}_0^{(0)})}{|\mathbf{E}_0|^2}.$$

The overlap of the two edges occurs for $\bar{s} \in I = [0, 1] \cap [s_0, s_1] \neq \emptyset$. The corresponding points in the contact set are $\mathbf{P}_0^{(0)} + T\mathbf{W}^{(0)} + \bar{s}\mathbf{E}_0^{(0)}$ for $\bar{s} \in I$.

In the event the two polygons are initially overlapping, the contact set is more expensive to construct. This set can be constructed by standard methods involving Boolean operations on polygons.

The pseudocode is shown next. The intersection is a convex polygon and is returned in the last two arguments of the function. If the intersection set is nonempty,

the return value of the function is true. The set must itself be convex. The number of vertices in the set is stored in quantity and the vertices, in counterclockwise order, are stored in the array I[]. If the return value is false, the last two arguments of the function are invalid and should not be used.

```
bool FindIntersection (ConvexPolygon C0, Vector V0,
    ConvexPolygon C1, Vector W1, double tmax, double& tfirst,
    double& tlast, int& quantity, Point I[])
{
    ProjInfo info0, info1, curr0, curr1;
    // Process as if C0 were stationary and C1 were moving.
    V = V1 - V0;
    tfirst = 0;
    tlast = INFINITY;

    // Process edges of C0.
    for (i0 = C0.GetN() - 1, i1 = 0; i1 < C0.GetN(); i0 = i1++)
    {
        D = C0.GetNormal(i0);
        ComputeInterval(C0,D,info0);
        ComputeInterval(C1,D,info1);
        speed = Dot(D,v);
        if (NoIntersect(tmax,speed,info0,info1,curr0,curr1,side,
            tfirst,tlast))
        {
            return false;
        }
    }

    // Process edges of C1.
    for (i0 = C1.GetN() - 1, i1 = 0; i1 < C1.GetN(); i0 = i1++)
    {
        D = C1.GetNormal(i0);
        ComputeInterval(C0,D,info0);
        ComputeInterval(C1,D,info1);
        speed = Dot(D,v);
        if (NoIntersect(tmax,speed,info0,info1,curr0,curr1,side,
            tfirst,tlast))
        {
            return false;
        }
    }

    // Compute the contact set.
    GetIntersection(C0,V0,C1,V1,curr0,curr1,side,tfirst,
```

```

        quantity,I);
    return true;
}

```

The intersection calculator pseudocode is shown next. Observe how the projection types are used to determine if the contact is vertex–vertex, edge–vertex, or edge–edge.

```

void GetIntersection (ConvexPolygon C0, Vector V0,
    ConvexPolygon C1, Vector V1, ProjInfo info0,
    ProjInfo info1, int side, double tfirst,
    int& quantity, Point I[])
{
    if (side == 1) // C0-max meets C1-min.
    {
        if (info0.isUnique[1])
        {
            // vertex-vertex or vertex-edge intersection
            quantity = 1;
            I[0] = C0.GetVertex(info0.index[1]) + tfirst * V0;
        }
        else if (info1.isUnique[0])
        {
            // vertex-vertex or edge-vertex intersection
            quantity = 1;
            I[0] = C1.GetVertex(info1.index[0]) + tfirst * V1;
        }
        else
        {
            // edge-edge intersection
            P = C0.GetVertex(info0.index[1]) + tfirst * V0;
            E = C0.GetEdge(info0.index[1]);
            Q0 = C1.GetVertex(info1.index[0]);
            Q1 = C1.GetVertex(info1.index[0] + 1);
            s0 = Dot(E,Q1-P) / Dot(E,E);
            s1 = Dot(E,Q0-P) / Dot(E,E);
            FindIntervalIntersection(0,1,s0,s1,quantity,
                interval);
            for (i = 0; i < quantity; i++)
            {
                I[i] = P + interval[i] * E;
            }
        }
    }
    else if (side == -1) // C1-max meets C0-min.

```

```

{
    if (info1.isUnique[1])
    {
        // vertex-vertex or vertex-edge intersection
        quantity = 1;
        I[0] = C1.GetVertex(info1.index[1]) + tfirst * V1;
    }
    else if (info0.isUnique[0])
    {
        // vertex-vertex or edge-vertex intersection
        quantity = 1;
        I[0] = C0.GetVertex(info0.index[0]) + tfirst * V0;
    }
    else
    {
        // edge-edge intersection
        P = C1.GetVertex(info1.index[1]) + tfirst * V1;
        E = C1.GetEdge(info1.index[1]);
        Q0 = C0.GetVertex(info0.index[0]);
        Q1 = C0.GetVertex(info0.index[0] + 1);
        s0 = Dot(E, Q1 - P) / Dot(E, E);
        s1 = Dot(E, Q0 - P) / Dot(E, E);
        FindIntervalIntersection(0, 1, s0, s1, quantity,
            interval);
        for (i = 0; i < quantity; i++)
        {
            I[i] = P + interval[i] * E;
        }
    }
}
else // Polygons were initially intersecting.
{
    ConvexPolygon COMoved = C0 + tfirst * V0;
    ConvexPolygon C1Moved = C1 + tfirst * V1;
    FindPolygonIntersection(COMoved, C1Moved, quantity, I);
}
}

```

The final case is the point at which the two polygons were initially overlapping so that the first time of contact is $T = 0$. FindPolygonIntersection is a general routine for computing the intersection of two polygons. In our collision detection system with the nonpenetration constraint, we should not need to worry about the last case, although you might want to trap the condition for debugging purposes.

Separation of Convex Polyhedra

The structure of the algorithm for convex polyhedra moving with constant linear velocity is similar to the one for convex polygons, except for the set of potential separating axes that must be tested. The pseudocode is

```
bool TestIntersection (ConvexPolyhedron C0, Vector V0,
    ConvexPolyhedron C1, Vector V1, double tmax,
    double& tfirst, double& tlast)
{
    // Process as if C0 were stationary and C1 were moving.
    V = V1 - V0;
    tfirst = 0;
    tlast = INFINITY;

    // Test faces of C0 for separation.
    for (i = 0; i < C0.GetFCount(); i++)
    {
        D = C0.GetNormal(i);
        ComputeInterval(C0,D,min0,max0);
        ComputeInterval(C1,D,min1,max1);
        speed = Dot(D,V);
        if (NoIntersect(tmax,speed,min0,max0,min1,max1,
            tfirst,tlast))
        {
            return false;
        }
    }

    // Test faces of C1 for separation.
    for (j = 0; j < C1.GetFCount(); j++)
    {
        D = C1.GetNormal(j);
        ComputeInterval(C0,D,min0,max0);
        ComputeInterval(C1,D,min1,max1);
        speed = Dot(D,V);
        if (NoIntersect(tmax,speed,min0,max0,min1,max1,
            tfirst,tlast))
        {
            return false;
        }
    }

    // Test cross products of pairs of edges.
```

```

for (i = 0; i < C0.GetECount(); i++)
{
    for (j = 0; j < C1.GetECount(); j++)
    {
        D = Cross(C0.GetEdge(i), C1.GetEdge(j));
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        speed = Dot(D, V);
        if (NoIntersect(tmax, speed, min0, max0, min1, max1,
                        tfirst, tlast))
        {
            return false;
        }
    }
}

return true;
}

```

The function `NoIntersect` is exactly the one used in the 2D problem.

Contact Set for Convex Polyhedra

The find-intersection query for two stationary convex polyhedra is a special example of Boolean operations on polyhedra. Since we are assuming nonpenetration in our collision system, we do not need to implement this.

Given two moving convex objects C_0 and C_1 , initially not intersecting, with velocities $\mathbf{V}_0$ and $\mathbf{V}_1$, if $T > 0$ is the first time of contact, the sets $C_0 + T\mathbf{V}_0 = \{\mathbf{X} + T\mathbf{V}_0 : \mathbf{X} \in C_0\}$ and $C_1 + T\mathbf{V}_1 = \{\mathbf{X} + T\mathbf{V}_1 : \mathbf{X} \in C_1\}$ are just touching with no interpenetration. As indicated earlier for convex polyhedra, the contact is one of face-face, face-edge, face-vertex, edge-edge, edge-vertex, or vertex-vertex. The analysis is slightly more complicated than that of the 2D setting, but the ideas are the same—the relative orientation of the convex polyhedra to each other must be known to properly compute the contact set.

The `TestIntersection` function can be modified to keep track of which vertices, edges, or faces are projected to the endpoints of the projection interval. At the first time of contact, this information is used to determine how the two objects are oriented with respect to each other. If the contact is vertex-vertex, vertex-edge, or vertex-face, then the contact point is a single point, a vertex. If the contact is edge-edge, the contact is typically a single point but can be an entire line segment. If the contact is edge-face, the contact set is a line segment. Finally, if the contact is face-face, the intersection set is a convex polygon. This is the most complicated scenario and requires a 2D convex polygon intersector. Each endpoint of the projection interval is either

generated by a vertex, an edge, or a face. Similar to the implementation for the 2D problem, a projection information class can be defined.

```
class ProjInfo
{
public:
    double min, max; // projection interval [min,max]
    int index[2];
    enum Type { V, E, F };
    Type type[2];
};
```

The zero-indexed values correspond to the minimum of the interval, the one-indexed values to the maximum. If the extreme point is exactly a vertex, the type is set to V. If the extreme points are exactly an edge, the type is set to E. If the extreme points are exactly a face, the type is set to F.

Just as for convex polygons, the extremal query must be modified to support calculation of the contact set via ProjInfo. In particular, we need to know the enumerated type to assign based on the extremal set.

```
int GetExtremeIndex (ConvexPolyhedron C, Vector D,
    ProjInfo::Type& type);
```

An implementation will need another interface function for ConvexPolyhedron that can determine the face bounded by the input edges.

```
class ConvexPolyhedron
{
public:
    // other members...

    int GetFaceFromEdges (array<int> edges);
};
```

Alternatively, you could build more information into the BSP tree nodes so that this information is immediately available. Function ComputeInterval must be modified to provide more information than just the projection interval.

```
void ComputeInterval (ConvexPolyhedron C, Vector D,
    ProjInfo& info)
{
    info.index[0] = GetExtremeIndex(C,-D,info.type[0]);
    info.min = Dot(D,C.GetVertex(info.index[0]));
    info.index[1] = GetExtremeIndex(C,+D,info.type[1]);
    info.max = Dot(D,C.GetVertex(info.index[1]));
}
```


The `NoIntersect` function that was modified in two dimensions to accept `ProjInfo` objects instead of projection intervals is used exactly as is for the 3D problem, so I do not restate that code here. With all modifications to this point, `TestIntersection` is rewritten as

```
bool TestIntersection (ConvexPolyhedron C0, Vector V0,
    ConvexPolyhedron C1, Vector V1, double tmax,
    double& tfirst, double& tlast)
{
    ProjInfo info0, info1, curr0, curr1;
    // Process as if C0 were stationary and C1 were moving.
    V = V1 - V0;
    tfirst = 0;
    tlast = INFINITY;

    // Test faces of C0 for separation.
    for (i = 0; i < C0.GetFCount(); i++)
    {
        D = C0.GetNormal(i);
        ComputeInterval(C0,D,info0);
        ComputeInterval(C1,D,info1);
        speed = Dot(D,V);
        if (NoIntersect(tmax,speed,info0,info1,curr0,curr1,
            side,tfirst,tlast))
        {
            return false;
        }
    }

    // Test faces of C1 for separation.
    for (j = 0; j < C1.GetFCount(); j++)
    {
        D = C1.GetNormal(j);
        ComputeInterval(C0,D,info0);
        ComputeInterval(C1,D,info1);
        speed = Dot(D,V);
        if (NoIntersect(tmax,speed,info0,info1,curr0,curr1,
            side,tfirst,tlast))
        {
            return false;
        }
    }

    // Test cross products of pairs of edges.
```

```

for (i = 0; i < C0.GetECount(); i++)
{
    for (j = 0; j < C1.GetECount(); j++)
    {
        D = Cross(C0.GetEdge(i), C1.GetEdge(j));
        ComputeInterval(C0, D, info0);
        ComputeInterval(C1, D, info1);
        speed = Dot(D, V);
        if (NoIntersect(tmax, speed, info0, info1, curr0,
                        curr1, side, tfirst, tlast))
        {
            return false;
        }
    }
}

return true;
}

```

The FindIntersection pseudocode has exactly the same implementation as TestIntersection, but with one additional block of code that is reached after all the loops if there will be an intersection. When the polyhedra intersect at time T , they are effectively moved with their respective velocities and the contact set is calculated. The pseudocode follows. The intersection is a convex polyhedron and is returned in the last argument of the function, but keep in mind that for nonpenetration, we should have only a convex polygon in 3D. If the intersection set is nonempty, the return value is true. Otherwise, the original moving convex polyhedra do not intersect and the function returns false.

```

bool FindIntersection (ConvexPolyhedron C0, Vector W0,
                      ConvexPolyhedron C1, Point W1, double tmax,
                      double& tfirst, double& tlast, ConvexPolyhedron& I)
{
    ProjInfo info0, info1, curr0, curr1;
    // Process as if C0 were stationary and C1 were moving.
    V = V1 - V0;
    tfirst = 0;
    tlast = INFINITY;

    // Test faces of C0 for separation.
    for (i = 0; i < C0.GetFCount(); i++)
    {
        D = C0.GetNormal(i);
        ComputeInterval(C0, D, info0);
    }
}

```

```

        ComputeInterval(C1,D,info1);
        speed = Dot(D,V);
        if (NoIntersect(tmax,speed,info0,info1,curr0,curr1,
                        side,tfirst,tlast))
        {
            return false;
        }
    }

    // Test faces of C1 for separation.
    for (j = 0; j < C1.GetFCount(); j++)
    {
        D = C1.GetNormal(j);
        ComputeInterval(C0,D,info0);
        ComputeInterval(C1,D,info1);
        speed = Dot(D,V);
        if (NoIntersect(tmax,speed,info0,info1,curr0,curr1,
                        side,tfirst,tlast))
        {
            return false;
        }
    }

    // Test cross products of pairs of edges.
    for (i = 0; i < C0.GetECount(); i++)
    {
        for (j = 0; j < C1.GetECount(); j++)
        {
            D = Cross(C0.GetEdge(i),C1.GetEdge(j));
            ComputeInterval(C0,D,info0);
            ComputeInterval(C1,D,info1);
            speed = Dot(D,V);
            if (NoIntersect(tmax,speed,info0,info1,curr0,
                            curr1,side,tfirst,tlast))
            {
                return false;
            }
        }
    }

    // Compute the contact set.
    GetIntersection(C0,V0,C1,V1,curr0,curr1,side,tfirst,I);
    return true;
}

```

The intersection calculator pseudocode follows.

```

void GetIntersection (ConvexPolyhedron C0, Vector V0,
    ConvexPolyhedron C1, Vector V1, ProjInfo info0,
    ProjInfo info1, int side, double tfirst,
    ConvexPolyhedron& I)
{
    if (side == 1) // C0-max meets C1-min.
    {
        if (info0.type[1] == ProjInfo::V)
        {
            // vertex-{vertex/edge/face} intersection
            I.InsertFeature(C0.GetVertex(info0.index[1]) +
                tfirst * V0);
        }
        else if (info1.type[0] == ProjInfo::V)
        {
            // {vertex/edge/face}-vertex intersection
            I.InsertFeature(C1.GetVertex(info1.index[0]) +
                tfirst * V1);
        }
        else if (info0.type[1] == ProjInfo::E)
        {
            Segment E0 = C0.GetESegment(info0.index[1]) +
                tfirst * V0;
            if (info1.type[0] == ProjInfo::E)
            {
                Segment E1 = C1.GetESegment(info1.index[0]) +
                    tfirst * V1;
                I.InsertFeature(IntersectSegmentSegment(E0,E1));
            }
            else
            {
                Polygon F1 = C1.GetFPolygon(info1.index[0]) +
                    tfirst * V1;
                I.InsertFeature(IntersectSegmentPolygon(E0,F1));
            }
        }
        else if (info1.type[0] == ProjInfo::E)
        {
            Segment E1 = C1.GetESegment(info1.index[0]) +
                tfirst * V1;
            if (info0.type[1] == ProjInfo::E)
            {

```

```

        Segment E0 = C0.GetESegment(info0.index[1]) +
            tfirst * V0;
        I.InsertFeature(IntersectSegmentSegment(E1,E0));
    }
    else
    {
        Polygon F0 = C0.GetFPolygon(info0.index[1]) +
            tfirst * V0;
        I.InsertFeature(IntersectSegmentPolygon(E1,F0));
    }
}
else // info0.type[1], info1.type[0] both ProjInfo::F
{
    // face-face intersection
    Polygon F0 = C0.GetFPolygon(info0.index[1]) +
        tfirst * V0;
    Polygon F1 = C1.GetFPolygon(info1.index[0]) +
        tfirst * V1;
    I.InsertFeature(IntersectPolygonPolygon(F0,F1));
}
}
else if (side == -1) // C1-max meets C0-min.
{
    if (info1.type[1] == ProjInfo::V)
    {
        // vertex-{vertex/edge/face} intersection
        I.InsertFeature(C1.GetVertex(info1.index[1]) +
            tfirst * V1);
    }
    else if (info0.type[0] == ProjInfo::V)
    {
        // {vertex/edge/face}-vertex intersection
        I.InsertFeature(C0.GetVertex(info0.index[0]) +
            tfirst * V0);
    }
    else if (info1.type[1] == ProjInfo::E)
    {
        Segment E1 = C1.GetESegment(info1.index[1]) +
            tfirst * V1;
        if (info0.type[0] == ProjInfo::E)
        {
            Segment E0 = C0.GetESegment(info0.index[0]) +
                tfirst * V0;

```

```

        I.InsertFeature(IntersectSegmentSegment(E1,E0));
    }
    else
    {
        Polygon F0 = C0.GetFPolygon(info0.index[0]) +
            tfirst * V0;
        I.InsertFeature(IntersectSegmentPolygon(E1,F0));
    }
}
else if (info0.type[0] == ProjInfo::E)
{
    Segment E0 = C0.GetESegment(info0.index[0]) +
        tfirst * V0;
    if (info1.type[1] == ProjInfo::E)
    {
        Segment E1 = C1.GetESegment(info1.index[1]) +
            tfirst * V1;
        I.InsertFeature(IntersectSegmentSegment(E0,E1));
    }
    else
    {
        Polygon F1 = C1.GetFPolygon(info1.index[1]) +
            tfirst * V1;
        I.InsertFeature(IntersectSegmentPolygon(E0,F1));
    }
}
else // info1.type[1], info0.type[0] both ProjInfo::F
{
    // face-face intersection
    Polygon F0 = C0.GetFPolygon(info0.index[0]) +
        tfirst * V0;
    Polygon F1 = C1.GetFPolygon(info1.index[1]) +
        tfirst * V1;
    I.InsertFeature(Intersection(F0,F1));
}
}
else // Polyhedra were initially intersecting.
{
    ConvexPolyhedron M0 = C0 + tfirst * V0;
    ConvexPolyhedron M1 = C1 + tfirst * V1;
    I = IntersectionPolyhedronPolyhedron(M0,M1);
}
}

```

The type `Segment` refers to a line segment and the type `Polygon` refers to a convex polygon in 3D. The various functions `Intersect<Type1><Type2>` are almost generic intersection calculators. I say “almost” meaning that you know the two objects *must* intersect since the separating axis results say so. Given that they intersect, you can optimize generic intersection calculators to obtain your results. The possible outputs from the intersection calculators are points, line segments, or convex polygons, referred to collectively as “features.” The class `ConvexPolyhedron` must support construction by allowing the user to insert any of these features. I simply used the name `InsertFeature` to cover all the cases (overloading of the function name, so to speak). The function `GetESegment` returns some representation of a line segment, for example, a pair of points. The calculation $S + t * V$ (where S is a `Segment`, V is a vector, and t is a floating-point number) requires the vector class to support a scalar-times-vector operation. Moreover, the expression requires addition to be defined for a `Segment` object and a `Vector` object. Similarly, `GetFPolygon` returns some representation of the convex polygon face, for example, an ordered array of points. The calculation $F + t * V$ requires addition to be defined for a `Polygon` object and a `Vector` object.

As you can see, this is the workhorse of the collision system, the geometric details of calculating intersections of line segments and convex polygons. You should expect that this is a likely candidate for the bottleneck in your collision system. For this reason you will see simplified systems such as [Bar01] where the contact set is reduced to a container of points. The preceding intersection calculator can be greatly optimized for such a system.

8.1.4 ORIENTED BOUNDING BOXES

So far we have discussed collision detection for convex polyhedra in general terms. A very common polyhedron used in applications is an *oriented bounding box*, the acronym *OBB* used for short. The term *box* is enough to describe the shape, but the modifier *bounding* applies when the box contains a more complex object and is used as a coarse measure of that portion of space the object occupies. The modifier *oriented* refers to the fact that the box axes are not necessarily aligned with the standard coordinate axes. Bounding boxes (and volumes) are typically used to minimize the work of the collision detection system by *not* processing pairs of objects, if you can cheaply determine that they will not intersect. A descriptive name for this process is *collision culling*, suggestive of the culling that a graphics engine does in order not to draw objects if you can cheaply determine that they are not visible.

An OBB is defined by a center point C that acts as the origin for a coordinate system whose orthonormal axis directions are U_i for $i = 0, 1, 2$. The directions are normal vectors to the faces of the OBB. The half-widths or *extents* of the box along the coordinate axes are $e_i > 0$ for $i = 0, 1, 2$. Figure 8.16 shows an OBB and the intersection of the coordinate axes with three faces of the box. The eight vertices of the OBB are of the form

$$P = C + \sigma_0 e_0 U_0 + \sigma_1 e_1 U_1 + \sigma_2 e_2 U_2$$

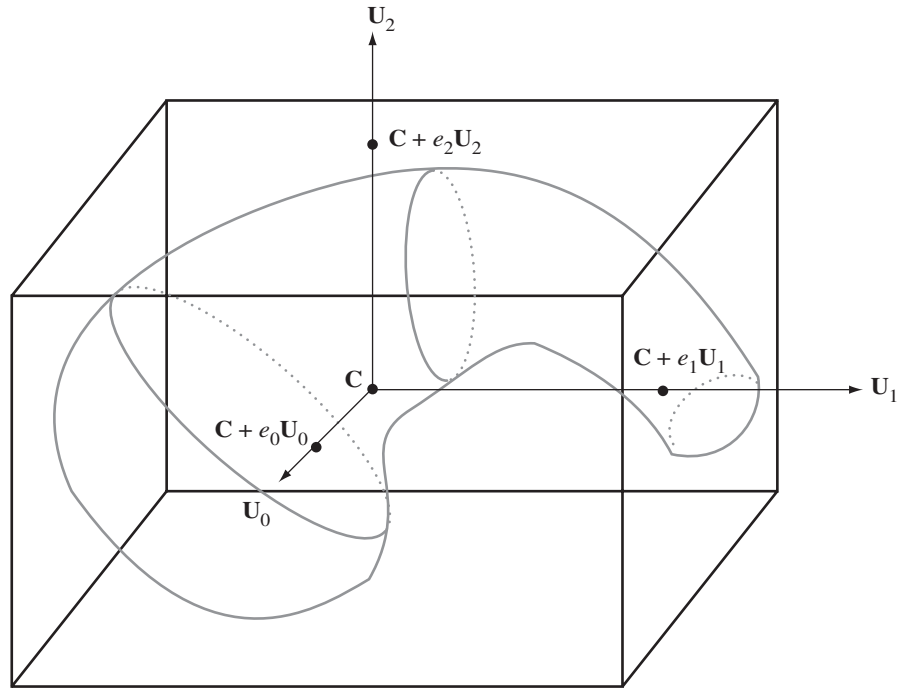


Figure 8.16 An OBB with center point C , coordinate axis directions U_0 , U_1 , and U_2 , and extents e_0 , e_1 , and e_2 along the coordinate axes. The object bounded by the box is outlined in gray.

where $|\sigma_i| = 1$ for $i = 0, 1, 2$; that is, we have eight choices for the signs σ_i .

Our interest is restricted to testing when two OBBs intersect, whether stationary or moving. As a convex polyhedron, an OBB has six faces and 12 edges. If we just blindly applied the test-intersection query for a pair of convex polyhedra, the number of potential separating axis tests is 156: six face normals for the first OBB, six face normals for the second OBB, and $144 = 12 * 12$ edge-edge pairs. In the worst case we would try all 156 axes only to find that the OBBs are intersecting. That is quite a large number of tests for such simple-looking objects! The nature of an OBB, though, is that the symmetry allows us to reduce the number of tests. You probably already observed that we have three pairs of parallel faces, so we only need to consider three face normals for an OBB for the purpose of separation. Similarly, only three edge directions are unique and happen to be those of the face normals. Thus, for a pair of OBBs we have only 15 potential separating axis tests: three face normals for the first OBB, three face normals for the second OBB, and $9 = 3 * 3$ edge-edge pairs.

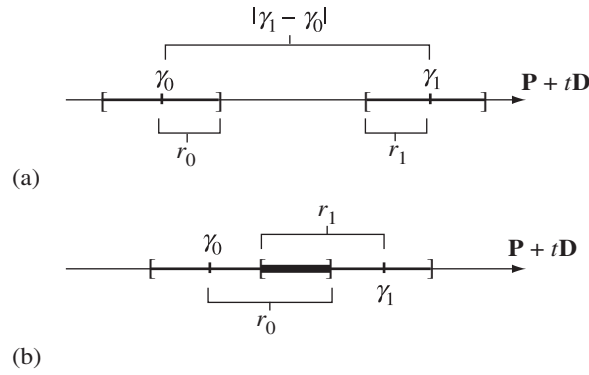


Figure 8.17 The projection intervals of two OBBs onto a line $\mathbf{P} + t\mathbf{D}$. (a) The intervals are disjoint, so the OBBs are separated. (b) The intervals overlap, so the line is not a separating axis.

We still have to project an OBB onto a potential separating axis $\mathbf{Q} + t\mathbf{D}$. Although the fast extremal query for general convex polyhedra may be applied, the symmetry of the OBB allows us to quickly determine the interval of projection. Since a vertex must be an extreme point, it suffices to try to find a vertex $\mathbf{P}$ that maximizes the dot product

$$\mathbf{D} \cdot (\mathbf{P} - \mathbf{Q}) = \mathbf{D} \cdot (\mathbf{C} - \mathbf{Q}) + e_0\sigma_0\mathbf{D} \cdot \mathbf{U}_0 + e_1\sigma_1\mathbf{D} \cdot \mathbf{U}_1 + e_2\sigma_2\mathbf{D} \cdot \mathbf{U}_2$$

The sign σ_0 is either 1 or -1 . To make the term $\sigma_0\mathbf{D} \cdot \mathbf{U}_0$ as large as possible, we want $\sigma_0 = 1$ when $\mathbf{D} \cdot \mathbf{U}_0 > 0$ and $\sigma_0 = -1$ when $\mathbf{D} \cdot \mathbf{U}_0 < 0$. If $\mathbf{D} \cdot \mathbf{U}_0 = 0$, it does not matter what the choice is for σ_0 . The resulting quantity can be written as the single term $|\mathbf{D} \cdot \mathbf{U}_0|$. The same argument applies to the other terms, so

$$\max \mathbf{D} \cdot (\mathbf{P} - \mathbf{Q}) = \mathbf{D} \cdot (\mathbf{C} - \mathbf{Q}) + e_0|\mathbf{D} \cdot \mathbf{U}_0| + e_1|\mathbf{D} \cdot \mathbf{U}_1| + e_2|\mathbf{D} \cdot \mathbf{U}_2|$$

Similarly, the minimum is

$$\min \mathbf{D} \cdot (\mathbf{P} - \mathbf{Q}) = \mathbf{D} \cdot (\mathbf{C} - \mathbf{Q}) - e_0|\mathbf{D} \cdot \mathbf{U}_0| - e_1|\mathbf{D} \cdot \mathbf{U}_1| - e_2|\mathbf{D} \cdot \mathbf{U}_2|$$

Therefore, the projection interval is $[\gamma - r, \gamma + r]$, where $\gamma = \mathbf{D} \cdot (\mathbf{C} - \mathbf{Q})$ and $r = \sum_{i=0}^2 e_i |\mathbf{D} \cdot \mathbf{U}_i|$.

Given two oriented bounding boxes, one with center $\mathbf{C}_0$, axes $\mathbf{A}_i$, and extents a_i , and one with center $\mathbf{C}_1$, axes $\mathbf{B}_i$, and extents b_i , let the projection intervals onto a line $\mathbf{Q} + t\mathbf{D}$ be $[\gamma_0 - r_0, \gamma_0 + r_0]$ and $[\gamma_1 - r_1, \gamma_1 + r_1]$. Figure 8.17 shows two cases, one with separated intervals and one with overlapping intervals.

The algebraic condition that describes the separated intervals in Figure 8.17 (a) is $|\gamma_1 - \gamma_0| > r_0 + r_1$. In words, this says that the distance between the centers of the projected intervals is larger than the sum of the radii of the intervals. The intervals in the bottom image of the figure overlap, so $|\gamma_1 - \gamma_0| < r_0 + r_1$. If the intervals are just touching, $|\gamma_1 - \gamma_0| = r_0 + r_1$. This last case is important when dealing with moving OBBs.

Define $r = |\gamma_1 - \gamma_0|$. A closer look at the algebraic condition for separation of the projected intervals shows that

$$r = |\gamma_1 - \gamma_0| = |\mathbf{D} \cdot (\mathbf{C}_1 - \mathbf{Q}) - \mathbf{D} \cdot (\mathbf{C}_0 - \mathbf{Q})| = |\mathbf{D} \cdot (\mathbf{C}_1 - \mathbf{C}_0)|$$

This means we need to specify only the direction $\mathbf{D}$ and not worry about providing a point $\mathbf{Q}$ on the line. Also,

$$r_0 = \sum_{i=0}^2 a_i |\mathbf{D} \cdot \mathbf{A}_i|, \quad r_1 = \sum_{i=0}^2 b_i |\mathbf{D} \cdot \mathbf{B}_i|$$

The condition for separation of the projection intervals is $r > r_0 + r_1$ and is formally expanded as

$$|\mathbf{D} \cdot \Delta| = r > r_0 + r_1 = \sum_{i=0}^2 a_i |\mathbf{D} \cdot \mathbf{A}_i| + \sum_{i=0}^2 b_i |\mathbf{D} \cdot \mathbf{B}_i| \quad (8.2)$$

where $\Delta = \mathbf{C}_1 - \mathbf{C}_0$. We have been thinking of $\mathbf{D}$ as a unit-length direction vector. The face normals are already unit-length potential separating directions. A potential separating direction $\mathbf{D} = \mathbf{A}_i \times \mathbf{B}_j$ obtained as a cross product of edges, one edge from each of the OBBs, is not necessarily unit length. We should then use $\mathbf{D}/|\mathbf{D}|$ in Equation (8.2) instead of $\mathbf{D}$. Notice, though, that the truth of the inequality is unchanged whether we use the vector or the normalized vector, since we can multiply through by $|\mathbf{D}|$. Consequently, we do not need to worry about normalizing the cross product.

A further optimization can be made. The formal sum for r_0 is a single term only when $\mathbf{D}$ is a face normal of the first OBB. For example, if $\mathbf{D} = \mathbf{A}_0$, then $r_0 = a_0$. The formal sum for r_1 is also a single term only when $\mathbf{D}$ is a face normal of the second OBB. The summation term of r_0 involves dot products $\mathbf{A}_i \cdot \mathbf{B}_j$ when using face normals of the second OBB for potential separating directions. The summation term of r_1 involves the same dot products when face normals of the first OBB are used for potential separating directions. When $\mathbf{D} = \mathbf{A}_j \times \mathbf{B}_k$, the summation term of r_0 is

$$\mathbf{D} \cdot \mathbf{A}_i = \mathbf{A}_j \times \mathbf{B}_k \cdot \mathbf{A}_i = \mathbf{A}_i \cdot \mathbf{A}_j \times \mathbf{B}_k = \mathbf{A}_i \times \mathbf{A}_j \cdot \mathbf{B}_k = \mathbf{A}_\ell \cdot \mathbf{B}_k$$

where $\{i, j, \ell\} = \{0, 1, 2\}$. For example, if $i = 2$ and $j = 0$, then $\ell = 1$. Similarly, the summation term of r_1 is

$$\mathbf{D} \cdot \mathbf{B}_i = \mathbf{A}_j \times \mathbf{B}_k \cdot \mathbf{B}_i = \mathbf{A}_j \cdot \mathbf{B}_k \times \mathbf{B}_i = \mathbf{A}_j \cdot \mathbf{B}_\ell$$

Table 8.2 Potential separating directions for OBBs and values for r_0 , r_1 , and r .

D	r_0	r_1	r
$\mathbf{A}_0$	a_0	$b_0 c_{00} + b_1 c_{01} + b_2 c_{02} $	$ \alpha_0 $
$\mathbf{A}_1$	a_1	$b_0 c_{10} + b_1 c_{11} + b_2 c_{12} $	$ \alpha_1 $
$\mathbf{A}_2$	a_2	$b_0 c_{20} + b_1 c_{21} + b_2 c_{22} $	$ \alpha_2 $
$\mathbf{B}_0$	$a_0 c_{00} + a_1 c_{10} + a_2 c_{20} $	b_0	$ \beta_0 $
$\mathbf{B}_1$	$a_0 c_{01} + a_1 c_{11} + a_2 c_{21} $	b_1	$ \beta_1 $
$\mathbf{B}_2$	$a_0 c_{02} + a_1 c_{12} + a_2 c_{22} $	b_2	$ \beta_2 $
$\mathbf{A}_0 \times \mathbf{B}_0$	$a_1 c_{20} + a_2 c_{10} $	$b_1 c_{02} + b_2 c_{01} $	$ c_{10}\alpha_2 - c_{20}\alpha_1 $
$\mathbf{A}_0 \times \mathbf{B}_1$	$a_1 c_{21} + a_2 c_{11} $	$b_0 c_{02} + b_2 c_{00} $	$ c_{11}\alpha_2 - c_{21}\alpha_1 $
$\mathbf{A}_0 \times \mathbf{B}_2$	$a_1 c_{22} + a_2 c_{12} $	$b_0 c_{01} + b_1 c_{00} $	$ c_{12}\alpha_2 - c_{22}\alpha_1 $
$\mathbf{A}_1 \times \mathbf{B}_0$	$a_0 c_{20} + a_2 c_{00} $	$b_1 c_{12} + b_2 c_{11} $	$ c_{20}\alpha_0 - c_{00}\alpha_2 $
$\mathbf{A}_1 \times \mathbf{B}_1$	$a_0 c_{21} + a_2 c_{01} $	$b_0 c_{12} + b_2 c_{10} $	$ c_{21}\alpha_0 - c_{01}\alpha_2 $
$\mathbf{A}_1 \times \mathbf{B}_2$	$a_0 c_{22} + a_2 c_{02} $	$b_0 c_{11} + b_1 c_{10} $	$ c_{22}\alpha_0 - c_{02}\alpha_2 $
$\mathbf{A}_2 \times \mathbf{B}_0$	$a_0 c_{10} + a_1 c_{00} $	$b_1 c_{22} + b_2 c_{21} $	$ c_{00}\alpha_1 - c_{10}\alpha_0 $
$\mathbf{A}_2 \times \mathbf{B}_1$	$a_0 c_{11} + a_1 c_{01} $	$b_0 c_{22} + b_2 c_{20} $	$ c_{01}\alpha_1 - c_{11}\alpha_0 $
$\mathbf{A}_2 \times \mathbf{B}_2$	$a_0 c_{12} + a_1 c_{02} $	$b_0 c_{21} + b_1 c_{20} $	$ c_{02}\alpha_1 - c_{12}\alpha_0 $

where again $\{i, j, \ell\} = \{0, 1, 2\}$. Therefore, all the separating axis tests require computing the quantities $c_{ij} = \mathbf{A}_i \cdot \mathbf{B}_j$ and do not need cross product operations. A convenient summary of the axes and quantities required by $r > r_0 + r_1$ is listed in Table 8.2. The table uses $\alpha_i = \Delta \cdot \mathbf{A}_i$ and $\beta_i = \Delta \cdot \mathbf{B}_i$. A term of the form $c_{10}\alpha_2 - c_{20}\alpha_1$ occurs as a result of $\Delta = \alpha_0\mathbf{A}_0 + \alpha_1\mathbf{A}_1 + \alpha_2\mathbf{A}_2$, and

$$\begin{aligned}
\mathbf{A}_0 \times \mathbf{B}_0 \cdot \Delta &= \alpha_0(\mathbf{A}_0 \times \mathbf{B}_0 \cdot \mathbf{A}_0) + \alpha_1(\mathbf{A}_0 \times \mathbf{B}_0 \cdot \mathbf{A}_1) + \alpha_2(\mathbf{A}_0 \times \mathbf{B}_0 \cdot \mathbf{A}_2) \\
&= \alpha_0(\mathbf{0}) - \alpha_1\mathbf{A}_2 \times \mathbf{B}_0 + \alpha_2\mathbf{A}_1 \times \mathbf{B}_0 \\
&= c_{10}\alpha_2 - c_{20}\alpha_1
\end{aligned}$$

Pseudocode follows. The code is organized to compute quantities only when needed. The code also detects when two face normals $\mathbf{A}_i$ and $\mathbf{B}_j$ are nearly parallel. Theoretically, if a parallel pair exists, it is sufficient to test only the face normals of the two OBBs for separation. Numerically, though, two nearly parallel faces can lead to all face normal tests reporting no separation along those directions. The cross product directions are tested next, but $\mathbf{A}_i \times \mathbf{B}_j$ is nearly the zero vector and can cause the system to report that the OBBs are not intersecting when in fact they are.

```

bool TestIntersection (OBB box0, OBB box1)
{
    // OBB: center C; axes U[0], U[1], U[2];
    //      extents e[0], e[1], e[2]

    // values that are computed only when needed
    double c[3][3];    // c[i][j] = Dot(box0.U[i],box1.U[j])
    double absC[3][3]; // |c[i][j]|
    double d[3];       // Dot(box1.C-box0.C,box0.U[i])

    // interval radii and distance between centers
    double r0, r1, r;
    int i;

    // cutoff for cosine of angles between box axes
    const double cutoff = 0.999999;
    bool existsParallelPair = false;

    // Compute difference of box centers.
    Vector diff = box1.C - box0.C;

    // axis C0 + t * A0
    for (i = 0; i < 3; i++)
    {
        c[0][i] = Dot(box0.U[0],box1.U[i]);
        absC[0][i] = |c[0][i]|;
        if (absC[0][i] > cutoff)
        {
            existsParallelPair = true;
        }
    }
    d[0] = Dot(diff,box0.U[0]);
    r = |d[0]|;
    r0 = box0.e[0];
    r1 = box1.e[0] * absC[0][0] + box1.e[1] * absC[0][1] +
        box1.e[2] * absC[0][2];
    if (r > r0 + r1)
    {
        return false;
    }

    // axis C0 + t * A1
    for (i = 0; i < 3; i++)
    {

```

```

        c[1][i] = Dot(box0.U[1],box1.U[i]);
        absC[1][i] = |c[1][i]|;
        if (absC[1][i] > cutoff)
        {
            existsParallelPair = true;
        }
    }
    d[1] = Dot(diff,box0.U[1]);
    r = |d[1]|;
    r0 = box0.e[1];
    r1 = box1.e[0] * absC[1][0] + box1.e[1] * absC[1][1] +
        box1.e[2] * absC[1][2];
    if (r > r0 + r1)
    {
        return false;
    }

    // axis C0 + t * A2
    for (i = 0; i < 3; i++)
    {
        c[2][i] = Dot(box0.U[2],box1.U[i]);
        absC[2][i] = |c[2][i]|;
        if (absC[2][i] > cutoff)
        {
            existsParallelPair = true;
        }
    }
    d[2] = Dot(diff,box0.U[2]);
    r = |d[2]|;
    r0 = box0.e[2];
    r1 = box1.e[0] * absC[2][0] + box1.e[1] * absC[2][1] +
        box1.e[2] * absC[2][2];
    if (r > r0 + r1)
    {
        return false;
    }

    // axis C0 + t * B0
    r = |Dot(diff,box1.U[0])|;
    r0 = box0.e[0] * absC[0][0] + box0.e[1] * absC[1][0] +
        box0.e[2] * absC[2][0];
    r1 = box1.e[0];
    if (r > r0 + r1)
    {

```

```

        return false;
    }

    // axis C0 + t * B1
    r = |Dot(diff, box1.U[1])|;
    r0 = box0.e[0] * absC[0][1] + box0.e[1] * absC[1][1] +
        box0.e[2] * absC[2][1];
    r1 = box1.e[1];
    if (r > r0 + r1)
    {
        return false;
    }

    // axis C0 + t * B2
    r = |Dot(diff, box1.U[2])|;
    r0 = box0.e[0] * absC[0][2] + box0.e[1] * absC[1][2] +
        box0.e[2] * absC[2][2];
    r1 = box1.e[2];
    if (r > r0 + r1)
    {
        return false;
    }

    if (existsParallelPair)
    {
        // A pair of box axes was (effectively) parallel, thus
        // boxes must intersect.
        return true;
    }

    // axis C0 + t * A0 x B0
    r = |d[2] * c[1][0] - d[1] * c[2][0]|;
    r0 = box0.e[1] * absC[2][0] + box0.e[2] * absC[1][0];
    r1 = box1.e[1] * absC[0][2] + box1.e[2] * absC[0][1];
    if (r > r0 + r1)
    {
        return false;
    }

    // Similar blocks of code go here for the remaining
    // axes C0 + t * A[i] x B[j].

    return true;
}

```

8.2 FINDING COLLISIONS BETWEEN MOVING OBJECTS

This section describes the mathematics for determining the first time of contact between two convex objects, each moving with a constant linear velocity. Although the paths of motion for the objects are usually not linear—the objects can have nonzero angular velocity—the paths may be decomposed into approximating subpaths. On each subpath the object is assumed to have constant linear velocity and zero angular velocity. The subpaths are a natural outcome of solving the differential equations of motion for the objects.

8.2.1 PSEUDODISTANCE

The algorithm presented here is general in that it requires only a *pseudodistance function* to be implemented for each pair of desired object types *and* only for the case of stationary objects. A pseudodistance function between two objects has the property that it is positive when the objects are separated, negative when the objects are overlapping, and zero when the objects are just touching. In the overlapping case, the volume of intersection is positive. In the just-touching case, the volume of intersection is zero. Such a function is usually something much easier to formulate than a true (signed) distance function for stationary objects or for any type of measurement between moving objects.

To illustrate in one dimension, consider two intervals $[p_0, p_1]$ and $[q_0, q_1]$. The intervals are separated when $p_1 < q_0$ or when $q_1 < p_0$. The intervals are just touching when $p_1 = q_0$ or when $q_1 = p_0$. Otherwise, the intervals are overlapping; for example, this is the case when $p_0 < q_0 < p_1 < q_1$. The length of intersection is $p_1 - q_0$. In the just-touching case, the length of intersection is zero since the intersection set is a single point. A pseudodistance function may be defined based on the separation axis tests. The two intervals are separated whenever the distance between their centers is larger than the sum of their radii. If the distance is equal to the sum of their radii, the intervals are just touching. If the distance is smaller, then the intervals overlap. The centers (midpoints) of the intervals are $(p_0 + p_1)/2$ and $(q_0 + q_1)/2$. The radii (half-widths) of the intervals are $(p_1 - p_0)/2$ and $(q_1 - q_0)/2$. Figure 8.18 shows a typical scenario of separation.

The separation test is

$$\left| \frac{p_0 + p_1}{2} - \frac{q_0 + q_1}{2} \right| > \frac{p_1 - p_0}{2} + \frac{q_1 - q_0}{2}$$

The difference of the two sides of the inequality makes for a reasonable pseudodistance function. However, we will remove the absolute value by squaring both sides first and subtract to produce the pseudodistance function

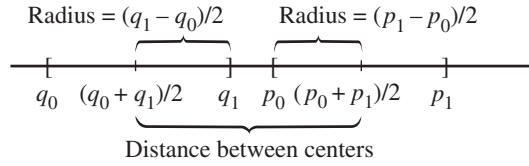


Figure 8.18 Two separated intervals.

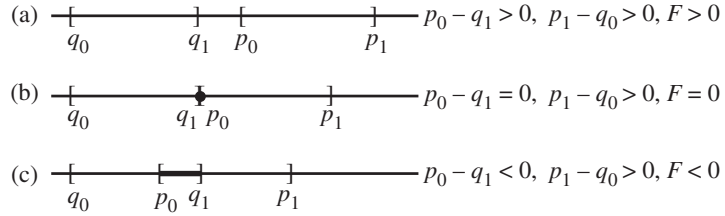


Figure 8.19 Three different interval configurations. (a) The intervals are separated. (b) The intervals are touching at a single point. (c) The intervals are overlapping.

$$\begin{aligned}
 F([p_0, p_1], [q_0, q_1]) &= \left(\frac{p_0 + p_1}{2} - \frac{q_0 + q_1}{2} \right)^2 - \left(\frac{p_1 - p_0}{2} + \frac{q_1 - q_0}{2} \right)^2 \\
 &= (p_0 - q_1)(p_1 - q_0)
 \end{aligned}$$

The squaring is representative of the 3D setting where we avoid square root calculations. Notice that the pseudodistance is independent of the order of the intervals:

$$\begin{aligned}
 F([q_0, q_1], [p_0, p_1]) &= (q_0 - p_1)(q_1 - p_0) = (p_0 - q_1)(p_1 - q_0) \\
 &= F([p_0, p_1], [q_0, q_1])
 \end{aligned}$$

Figure 8.19 shows three different interval configurations; Figure 8.19 (a) shows separated intervals ($F > 0$); Figure 8.19 (b) shows just-touching intervals ($F = 0$); and Figure 8.19 (c) shows overlapping intervals ($F < 0$).

8.2.2 CONTACT BETWEEN MOVING INTERVALS

The pseudodistance function F may be used to determine whether two stationary intervals are separated, just touching, or overlapping. But what if the intervals are moving with constant speeds? Let the interval $[p_0, p_1]$ move with velocity u (a signed scalar; speed is $|u|$) and let the interval $[q_0, q_1]$ move with velocity v (a signed scalar; speed is $|v|$). The problem is to compute the *first contact time*, $t_{\text{first}} \geq 0$, and the *first contact point*, c_{first} , between the moving intervals. The subscripts *first* are indicative of computing the first such time and point. However, the intervals can pass through each other, leading to a *last contact time*, t_{last} , and a *last contact point*, c_{last} . Although you would think the last time and point are irrelevant in the intersection query, it turns out that they are relevant for 3D queries involving the method of separating axes.

The moving intervals are $[p_0 + tu, p_1 + tu]$ and $[q_0 + tv, q_1 + tv]$. The pseudodistance function may be used to determine the contact time; that is,

$$\begin{aligned} G(t) &= F([p_0 + tu, p_1 + tu], [q_0 + tv, q_1 + tv]) \\ &= [(p_0 + tu) - (q_1 + tv)][(p_1 + tu) - (q_0 + tv)] \\ &= (u - v)^2 t^2 + 2(u - v)[(p_0 + p_1)/2 - (q_1 + q_0)/2]t + (p_0 - q_1)(p_1 - q_0) \\ &= a_2 t^2 + 2a_1 t + a_0 \end{aligned}$$

which is a quadratic function of time when the relative velocity $(u - v)$ is not zero. Notice that a_0 is the pseudodistance between the intervals at time zero. The value a_1 is the relative velocity of the intervals multiplied by the signed distance between interval centers. The value $a_2 \geq 0$. When $a_2 > 0$, the graph of $G(t)$ is a parabola that opens upward. The determination of the contact time $T \geq 0$ amounts to computing the roots of $G(t)$, but we do so by appealing to the structure of the graph of $G(t)$.

Intervals Initially Separated

The common scenario is that the two intervals are initially separated, in which case $G(0) > 0$. Figure 8.20 shows four possible cases for the graph of G . Figure 8.20 (a) shows the case of $a_1 \geq 0$. The graph does not reach the t -axis. In fact, $G'(t) = 2(a_2 t + a_1) \geq 0$ for $t \geq 0$, so the function is increasing for all time $t \geq 0$. The coefficient is

$$a_1 = (u - v) \left(\frac{p_0 + p_1}{2} - \frac{q_0 + q_1}{2} \right)$$

For this to be nonnegative, the signs of $(u - v)$ and $((p_0 + p_1)/2 - (q_1 + q_0)/2)$ must be equal. To see what this means physically, assume that $[p_0, p_1]$ is to the right of $[q_0, q_1]$ initially. The signed difference of centers is $((p_0 + p_1)/2 - (q_1 + q_0)/2) > 0$. If $(u - v) > 0$, then the intervals are *moving apart* and can never intersect. The graph

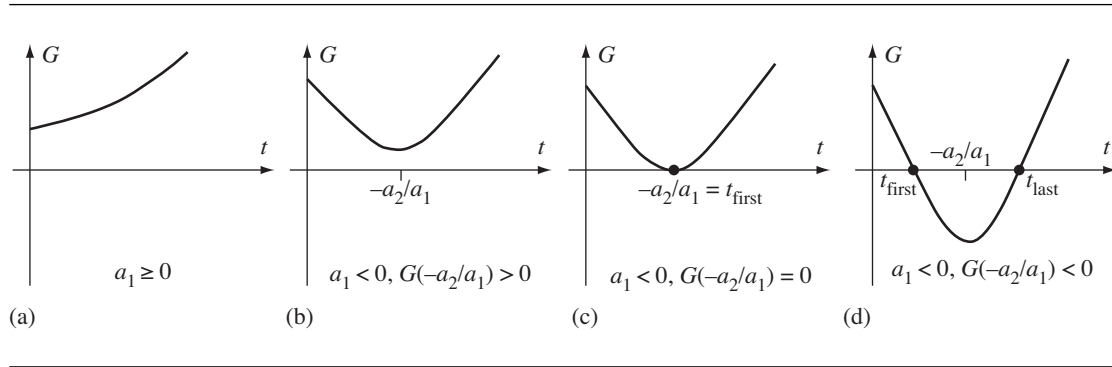


Figure 8.20 The four possible graphs of G when the intervals are initially separated.

of G in Figure 8.20 (a) confirms this. Similarly, if $[p_0, p_1]$ is to the left of $[q_0, q_1]$ initially, then the signed difference of centers is $((p_0 + p_1)/2 - (q_1 + q_0)/2) < 0$. If $(u - v) < 0$, then the intervals are moving apart and can never intersect.

It turns out that the graphs shown in Figure 8.20 (b) and (c) cannot occur! Define $r = u - v$, $\delta = ((p_0 + p_1)/2 - (q_1 + q_0)/2)$, and $\rho = (p_0 - q_1)(p_1 - q_0)$. The quadratic equation has roots

$$t = \frac{-a_1 \pm \sqrt{a_1^2 - a_0 a_2}}{a_2} = \frac{-\delta \pm \sqrt{\delta^2 - \rho}}{r}$$

The discriminant is

$$\delta^2 - \rho = \frac{1}{4} \left((p_1 - p_0)^2 + (q_1 - q_0)^2 \right) > 0$$

Positivity of the discriminant is guaranteed when the intervals are not degenerate; that is, when $p_1 > p_0$ and $q_1 > q_0$. This means that the quadratic equation always has two distinct real-valued roots whenever $a_1 < 0$. Physically, this makes sense. When $a_1 < 0$, the intervals are *moving toward each other* with constant speed. They must eventually intersect for the first time, overlap for some time while they pass through each other, intersect for the last time, and then remain separated for all times after the last contact. The graph shown in Figure 8.20 (d) is the only other possibility. In 2D and 3D, all four graphs shown in Figure 8.20 are possible for the convex objects we deal with. In 1D, though, the confinement to a line makes it difficult for one interval to skirt around the other without contact!

Using the quadratic formula in this case is apparently overkill. Just from the perspective of the two intervals moving together, the first and last contact times are solutions to linear equations. When $[p_0, p_1]$ is to the right of $[q_0, q_1]$ initially, and

if the intervals are moving toward each other, the first time of contact occurs when $p_0 + tu = q_1 + tv$, so

$$t_{\text{first}} = \frac{q_1 - p_0}{u - v}, \quad c_{\text{first}} = p_0 + t_{\text{first}}u$$

When $[p_0, p_1]$ is to the left of $[q_0, q_1]$ initially, and if the intervals are moving toward each other, the first time of contact occurs when $p_1 + tu = q_0 + tv$, so

$$t_{\text{first}} = \frac{q_0 - p_1}{u - v}, \quad c_{\text{first}} = p_0 + t_{\text{first}}u$$

The reason for introducing the quadratic approach is that generally the 3D cases are not linear, and the pseudodistance functions are not trivial. In the next section we will see how to compute the roots of $G(t)$ using numerical methods.

Intervals Initially Overlapping

It is possible that the intervals $[p_0, p_1]$ and $[q_0, q_1]$ are initially just touching or overlapping. In this case you may as well report that $t_{\text{first}} = 0$. The problem is deciding how to report the contact set, which might consist of a full interval of points itself. This is a judgment call to be made in the design of any collision detection system. Computing the full contact set can require more cycles than an application is willing to commit to. The alternative is to report one point that is in the contact set. You could choose an endpoint of one of the intervals, or perhaps the midpoint of the contact set.

8.2.3 COMPUTING THE FIRST TIME OF CONTACT

Consider the graph of $G(t)$ in Figure 8.20 (d). Starting at time $t = 0$, we would like to search through future times to locate the first contact time t_{first} . Newton's method may be used to do this. Figure 8.21 shows the geometric setup. Given the current root estimate t_k , the next estimate is computed as the intersection of the tangent line to the graph at $(t_k, G(t_k))$ with the t -axis. The slope of the graph is $G'(t_k)$ and the tangent line is $G - G(t_k) = G'(t_k)(t - t_k)$. Setting $G = 0$, the intersection is determined by $0 - G(t_k) = G'(t_k)(t_{k+1} - t_k)$. Thus,

$$t_{k+1} = t_k - \frac{G(t_k)}{G'(t_k)}, \quad t_0 = 0$$

From the figure, notice that $G'(t_k) < 0$ and $G(t_k) > 0$, which implies $t_{k+1} > t_k$. This is true no matter how many iterations you have computed. Consequently, the sequence $\{t_k\}_{k=0}^{\infty}$ is increasing. From the geometry, the iterates are always to the left of the root t_{first} , which means the iterates are bounded. From calculus, any increasing bounded

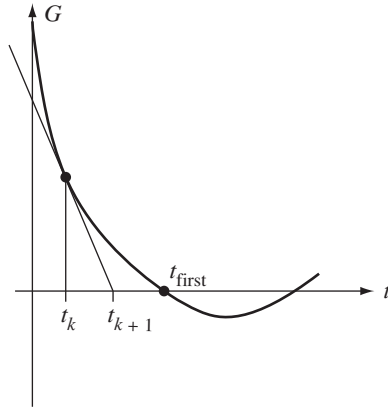


Figure 8.21 Newton's method to locate the first root of $G(t)$.

sequence must converge to a limit. In our case, that limit is the root. From a numerical perspective, what is important is how many iterations must be computed to obtain a root estimate t_k that is acceptable for the application. Collision detection needs to be fast, so the fewer the iterates the better. Collision detection needs to be accurate, which might require more iterates. A reasonable system needs to balance speed with accuracy, so the decision of when to stop iterating is important. Ideally, we would like $|t_k - t_{\text{first}}|$ to be very small *and* we would like $|G(t_k)|$ to be very small. Before we investigate this further, let's revisit the comment "From the geometry . . ."

The graph of $G(t)$ for the 1D problem of moving interval intersection is a parabola that opens upward. No matter which tangent line you look at, the graph is always above the line. A general class of functions having this property are *convex functions*. Such a function $G(t)$ has the property that

$$G((1-s)t_0 + st_1) \leq (1-s)G(t_0) + sG(t_1), \quad s \in [0, 1]$$

What this says geometrically is that the line segment connecting the graph points $(t_0, G(t_0))$ and $(t_1, G(t_1))$ is always above the graph of the function. Convex functions are ideal for root finding because the Newton iterates, when starting to the left of the first root, are increasing and guaranteed to converge to the root. Numerical analysis texts discuss this property, but only locally. That is, as long as $G''(r) < 0$ at a root r and as long as your iterates start near the root, you are guaranteed convergence. The nice thing about convex functions is that *you do not have to start near the root*.

The predictive collision detection that I am postulating here assumes that the objects are moving with constant linear velocity. Moreover, all the objects of interest

are convex sets. These facts guarantee that you can always build a time-varying pseudodistance function that is a convex function. Consequently, the methods described in this document allow you to build a predictive collision detection algorithm for any pair of convex objects. All you need to provide are the pseudodistance functions for the stationary objects.

The pseudocode for the algorithm is listed next. Some testing occurs initially in order to quickly exit the function when there will be no intersection. The code assumes the existence of a pseudodistance function and its derivative for the pair of object types. The code also maintains an enumeration for the type of contact (overlapping, just touching, separated).

```
// time-varying pseudodistance G(t)
float G (Object obj0, Point vel0, Object obj1, Point vel1,
        float t)
{
    // Let obj0 and obj1 denote sets of points. The
    // "moved sets" are
    //   obj0 + t * vel0 = {x + t * vel0 : x is a point in obj0}
    //   obj1 + t * vel1 = {x + t * vel1 : x is a point in obj1}
    // Compute the pseudodistance for the stationary objects.
    return Pseudodistance(obj0+t*vel0,obj1+t*vel1);
}

// derivative G'(t)
float GDer (Object obj0, Point vel0, Object obj1, Point vel1,
           float t)
{
    // Compute the derivative of the G(obj0,vel0,obj1,vel1,t)
    // function.
    return the derivative;
}

ContactType Find (Object obj0, Point vel0, Object obj1,
                 Point vel1, float tmax, float& contactTime)
{
    float t0 = 0;
    float g0 = G(obj0,vel0,obj1,vel1,t0);

    if (g0 < 0)
    {
        // Objects are overlapping.
        contactTime = 0;
        return OVERLAPPING;
    }
}
```

```

if (g0 == 0)
{
    // Objects are just touching.
    contactTime = 0;
    return TOUCHING;
}

// The objects are initially separated.
if (vel0 == vel1)
{
    // Relative velocity is zero; separated objects remain
    // separated.
    contactTime = -INFINITY;
    return SEPARATED;
}

float gder0 = GDer(obj0,vel0,obj1,vel1,t0);
if (gder0 >= 0)
{
    // Objects are moving apart.
    contactTime = -INFINITY;
    return SEPARATED;
}

float g1 = G(obj0,vel0,obj1,vel1,tmax);
if (g1 > 0)
{
    // Objects are separated at the maximum time.
    float gder1 = GDer(obj0,vel0,obj1,vel1,tmax);
    if (gder1 < 0)
    {
        // The objects are moving toward each other but
        // do not intersect during the time interval
        // [0,tmax]. If there is a first time of contact,
        // it must be that tmax < tfirst.
        contactTime = INFINITY;
        return SEPARATED;
    }
}

// We know  $[G(0) > 0 \text{ and } G'(0) < 0]$  and  $[G(tmax) \leq 0 \text{ or}$ 
//  $[G(tmax) > 0 \text{ and } G'(tmax) \geq 0]]$ . These conditions
// guarantee that the convex function  $G(t)$  has a root on
// the time interval  $[0,tmax]$ .

```

```

do forever
{
    t0 -= g0/gder0;
    g0 = G(obj0,vel0,obj1,vel1,t0);
    if (convergence criterion satisfied)
    {
        contactTime = t0;
        return TOUCHING;
    }
}
}

```

Understanding the pseudocode for Find is a matter of interpreting each code block relative to the graph of the convex function shown in Figure 8.21. The practical issues that must be dealt with are listed next.

1. Naturally, floating-point arithmetic causes us to pay attention to the comparisons of the floating-point values to zero. Small threshold values must be chosen to account for round-off errors.
2. The do forever loop is potentially infinite. In practice, the number of iterations is clamped to a maximum. If the convergence criterion is not satisfied for the maximum number of iterations, the loop terminates and you have to decide what response to give in the query. Because of the structure of the pseudocode, the response should be that a collision has occurred. The most common reason for failing to meet the convergence criterion is that the function values $G(t)$ are nearly zero near the root, but not close enough to zero to satisfy the convergence criterion, and the derivative values $G'(t)$ are very large near the root. The ratio $G(t)/G'(t)$ is very small, so the update on t is relatively insignificant and might not help to reduce $G(t)$ to be small enough to satisfy the convergence criterion.
3. The Find function should be given a chance to compute a contact point (or the full contact set, if desired). Whenever the function returns OVERLAPPING or JUST_TOUCHING, another function can be called to allow contact set computation.
4. It might not be possible to easily formulate a computable expression for the derivative of the pseudodistance function. Instead, a finite difference method can be chosen to estimate the derivative. The result will be that Newton's method is replaced by the Secant method. In theory, Newton's method is quadratically convergent (order 2) and the Secant method has an order of convergence equal to the golden ratio $(1 + \sqrt{5})/2 \doteq 1.618$. Should you choose to use finite differences, another parameter introduced into the system is the time step for the approximation.

8.2.4 ESTIMATING THE FIRST DERIVATIVE

There are many ways to estimate a derivative using finite differences. The most common low-order approximations are the forward difference,

$$G(t) = \frac{G(t+h) - G(t)}{h} + O(h)$$

the backward difference,

$$G(t) = \frac{G(t) - G(t-h)}{h} + O(h)$$

and the centered difference,

$$G(t) = \frac{G(t+h) - G(t-h)}{h} + O(h^2)$$

The application to the problem at hand should balance speed with accuracy. Given a t value and an already computed $G(t)$, a centered difference requires two more function evaluations for the specified step size $h > 0$. Rather than pay for two evaluations for $O(h^2)$ error, we will choose one of the other estimates for a cost of one evaluation, but only $O(h)$ error.

Figure 8.22 shows the iterations based on backward and forward differences and on the actual tangent line. The graph of $G(t)$ is drawn in black. The current iterate is t_0 . The next iterate from the tangent line (Newton's method) is t_1^n . The tangent line is drawn in red. The backward difference uses the secant line connecting $(t_0 - h, G(t_0 - h))$ and $(t_0, G(t_0))$, drawn in blue. The intersection of this line with the t -axis produces the next iterate t_1^b . The forward difference uses the secant line connecting $(t_0, G(t_0))$ and $(t_0 + h, G(t_0 + h))$, drawn in taupe. The intersection of this line with the t -axis produces the next iterate t_1^f .

The convexity of $G(t)$ guarantees that

$$t_0 - h < t_0 < t_1^b \leq t_1^n \leq t_1^f \leq t_{\text{first}}$$

The backward difference scheme generally is more conservative about the next iterate than Newton's method, but we are guaranteed that the iterates are monotonically increasing, just like the Newton iterates. Figure 8.22 shows the best behavior for the forward difference scheme—it produces an iterate larger than the Newton iterate, but smaller than the first time of contact. However, other situations can occur that require a forward difference approach to do more work to converge to the root.

Figure 8.23 shows various possibilities for the next iterate produced by the forward difference scheme. The distinctions between the cases and potential actions taken are listed next.

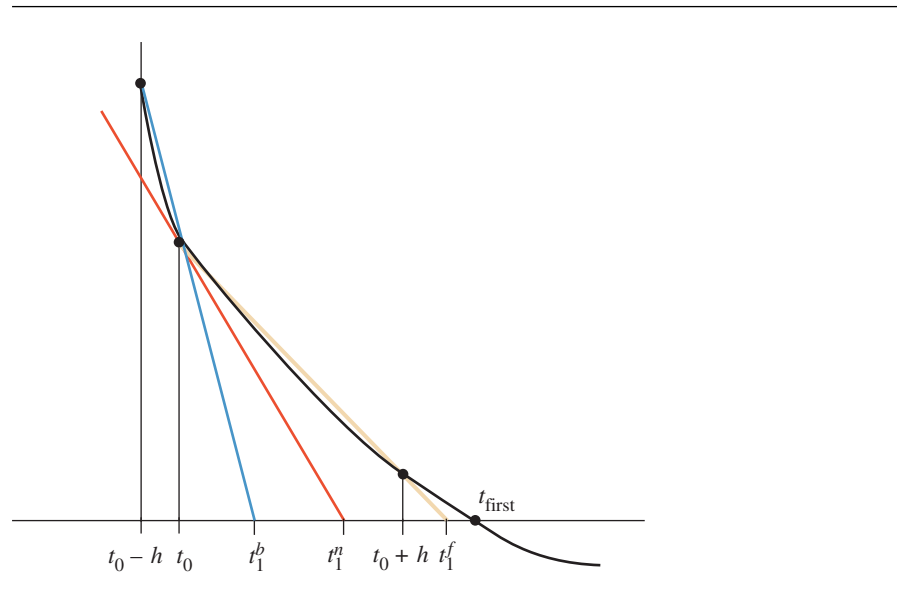


Figure 8.22 Choices of next iterate based on backward differences, forward differences, and the tangent line.

- (a) This is the best of all possibilities. The iterate satisfies $t_1 < t_{\text{first}}$ and is a better estimate of the root than what Newton's method produced. Notice that $G(t_1) > 0$ and $G'(t_1) < 0$.
- (b) The iterate exceeded the first time of contact but not the last time of contact, so $t_{\text{first}} < t_1 < t_{\text{last}}$. Notice that $G(t_0 + h) < 0$ and $G(t_1) < 0$. Bisection is applied while the iterates produce negative G values. Once a positive value is reached, another forward difference step is applied. This is a hybrid scheme that is applied until the convergence criteria are met. Alternatively, if a negative G value is selected, the time step h for the forward difference may be reduced in size in an attempt to produce a next iterate for which G is positive.
- (c) The iterate exceeded the last time of contact, so $t_{\text{last}} < t_1$ and $G(t_1) > 0$. Also in this case, the slope of the secant line is negative. Bisection may also be applied here, but always to find the first time of contact. That is, if you have two iterates t_k and t_{k+1} for which $G(t_k) > 0$ and $G(t_{k+1}) < 0$, the bisection is applied on that interval. But if $G(t_k) < 0$ and $G(t_{k+1}) > 0$, you do not apply the bisection since you have bounded the last time of contact.
- (d) The iterate slope of the secant line connecting $(t_0, G(t_0))$ and $(t_0 + h, G(t_0 + h))$ is positive. The intersection of this line with the t -axis produces the next iterate

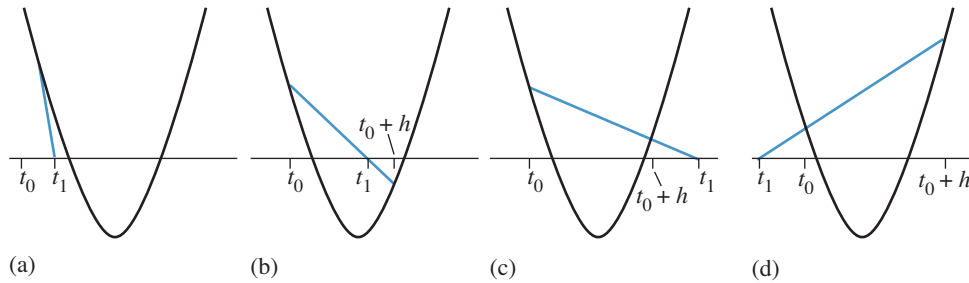


Figure 8.23 Choices of next iterate based on forward differences.

that is smaller than the previous one: $t_1 < t_0$. A smaller step size h should be chosen.

Trapping all the possible cases and handling them correctly will, in the worst case, lead to a bisection approach whose order of convergence is not quadratic. It is better to use the backward difference approach. The code simplicity is also attractive with this approach.

8.3 A DYNAMIC COLLISION DETECTION SYSTEM

Using an object-oriented design, the discussion of the last section allows us to create an abstract base class that represents the mechanism for computing the first contact time of two moving convex objects. Specific pairs of object types are handled by classes derived from the abstract base class.

8.3.1 THE ABSTRACT BASE CLASS

The abstract base class has the following interface.

```
class Colliders
{
public:
    Colliders ();
    virtual ~Colliders ();

    void SetDerivativeTimeStep (float fTimeStep);
    float GetDerivativeTimeStep () const;
```

```

void SetPseudodistanceThreshold (float fThreshold);
float GetPseudodistanceThreshold () const;
void SetMaxIterations (int iMaxIterations);
int GetMaxIterations () const;

enum CollisionType
{
    UNKNOWN,
    SEPARATED,
    TOUCHING,
    OVERLAPPING
};

virtual CollisionType Test (
    float fMaxTime,
    const Vector3f& rkVelocity0,
    const Vector3f& rkVelocity1,
    float& rfContactTime);

virtual CollisionType Find (
    float fMaxTime,
    const Vector3f& rkVelocity0,
    const Vector3f& rkVelocity1,
    float& rfContactTime);

protected:
    virtual float Pseudodistance (
        float fTime,
        const Vector3f& rkVelocity0,
        const Vector3f& rkVelocity1) const = 0;

    virtual void ComputeContactInformation (
        CollisionType eCollisionType,
        float fTime,
        const Vector3f& rkVelocity0,
        const Vector3f& rkVelocity1) const;

    virtual float PseudodistanceDerivative (
        float fT0, float fF0,
        const Vector3f& rkVelocity0,
        const Vector3f& rkVelocity1) const;

    virtual CollisionType FastNoIntersection (
        float fMaxTime,

```

```

    const Vector3f& rkVelocity0,
    const Vector3f& rkVelocity1,
    float& rff0, float& rffDer0) const;

    float m_fDerivativeTimeStep;
    float m_fInvDerivativeTimeStep;
    float m_fPseudodistanceThreshold;
    int m_iMaxIterations;
};

```

This is an abstract base class that implements the numerical root finder discussed in the previous section. The member `m_fDerivativeTimeStep` is the value h used in the backward difference estimate of the derivative. The member `m_fPseudodistanceThreshold` is used to determine when $G(t)$ is sufficiently close to zero (the convergence criterion). The member `m_iMaxIterations` is the maximum number of Newton's iterations that the solver will use.

The test-intersection and find-intersection queries are the class member functions `Test` and `Find`, respectively. The time interval for the intersection query is $[0, t_{\max}]$. The input parameter `fMaxTime` is the value $t_{\max}$. The two vector inputs are the constant linear velocities of the object during the specified time interval. The contact time is returned via the last parameter. The return value of the function indicates what will happen during the specified time interval. The output contact time is valid whenever the function's return value is `TOUCHING`. If this time is t_{first} , you are guaranteed that $0 \leq t_{\text{first}} \leq t_{\max}$. If the objects are initially overlapping, the function returns `OVERLAPPING` and a contact time of zero. If the function returns `SEPARATED`, the two objects will not collide during the specified time interval, in which case the contact time is invalid (actually set to $-\infty$). The base class `Test` function just calls the `Find` function. The `Test` and `Find` functions are virtual. The intent is that if your specific pair of object types lends itself to faster intersection queries, taking advantage of the special structure of the objects, then you can override the root-finding system.

Each derived class represents a pair of object types. The class must implement the `Pseudodistance` function for the two object types. The base class function `ComputeContactInformation` does nothing, but is called by the `Find` function. Your derived classes can compute the contact set (and contact normals) if so desired. Since the structure of these sets is specific to the object types, the derived classes must also provide accessors to these sets.

The function `FastNoIntersection` encapsulates the first portion of the root-finding system; namely, it checks to see if initially the objects are overlapping, just touching, separated and stationary relative to each other, or separated and moving away from each other. This function is also virtual. The intent is that if your specific pair of object types lends itself to faster no-intersection queries ("quick outs"), taking advantage of the special structure of the objects, then you can override the base class system. The base class implementation is

```

Colliders::CollisionType Colliders::FastNoIntersection (
    float fMaxTime, const Vector3f& rkVelocity0,
    const Vector3f& rkVelocity1, float& rff0,
    float& rffDer0) const
{
    // Analyze the initial configuration of the objects.
    rff0 = Pseudodistance(0.0f, rkVelocity0, rkVelocity1);

    if (rff0 <= -m_fPseudodistanceThreshold)
    {
        // Objects are (significantly) overlapping.
        ComputeContactInformation(OVERLAPPING, 0.0f,
            rkVelocity0, rkVelocity1);
        return OVERLAPPING;
    }

    if (rff0 <= m_fPseudodistanceThreshold)
    {
        // Objects are (nearly) in tangential contact.
        ComputeContactInformation(TOUCHING, 0.0f,
            rkVelocity0, rkVelocity1);
        return TOUCHING;
    }

    // The objects are not currently in contact or
    // overlapping. If the relative velocity between them is
    // zero, they cannot intersect at a later time.
    if (rkVelocity0 == rkVelocity1)
    {
        return SEPARATED;
    }

    // Compute or estimate the derivative  $F'(0)$ .
    rffDer0 = PseudodistanceDerivative(0.0f, rff0,
        rkVelocity0, rkVelocity1);
    if (rffDer0 >= 0.0f)
    {
        // The objects are moving apart.
        return SEPARATED;
    }

    // Check if the objects are not intersecting, yet still
    // moving toward each other at maximum time. If this
    // is the case, the objects do not intersect on the

```

```

// specified time interval.
float fF1 = Pseudodistance(fMaxTime,rkVelocity0,
    rkVelocity1);
if (fF1 > 0.0f)
{
    // Compute or estimate the derivative F'(tmax).
    float fFDer1 = PseudodistanceDerivative(
        fMaxTime,fF1,rkVelocity0,rkVelocity1);
    if (fFDer1 < 0.0f)
    {
        // The objects are moving toward each other and
        // do not intersect during the specified time
        // interval.
        return SEPARATED;
    }
}

return UNKNOWN;
}

```

The base class function Find is the following.

```

Colliders::CollisionType Colliders::Find (float fMaxTime,
    const Vector3f& rkVelocity0, const Vector3f& rkVelocity1,
    float& rfContactTime)
{
    float fF0, fFDer0;
    CollisionType eCollisionType = FastNoIntersection(
        fMaxTime,rkVelocity0,rkVelocity1,fF0,fFDer0);
    if (eCollisionType != UNKNOWN)
    {
        return eCollisionType;
    }

    // Use Newton's method for root finding when the derivative
    // is calculated but Secant method when the derivative
    // is estimated.
    float fT0 = 0.0f;
    for (int i = 1; i <= m_iMaxIterations; i++)
    {
        float fT0 -= fF0/fFDer0;

        // In theory, the iterates are smaller than the first
        // time of contact and the contact time is shorter
    }
}

```

```

// than the maximum time, so this conditional block
// will not be entered. However, there is a small
// chance you can enter the block because of numerical
// round-off errors when the contact time and maximum
// time are nearly the same. Entering the block
// effectively means the iterates have converged.
if (fT0 > fMaxTime)
{
    // The objects do not intersect during the
    // specified time interval.
    rfContactTime = fMaxTime;
    ComputeContactInformation(TOUCHING,rfContactTime,
        rkVelocity0,rkVelocity1);
    return TOUCHING;
}

fF0 = Pseudodistance(fT0,rkVelocity0,rkVelocity1);
if (fF0 <= m_fPseudodistanceThreshold)
{
    rfContactTime = fT0;
    ComputeContactInformation(TOUCHING,rfContactTime,
        rkVelocity0,rkVelocity1);
    return TOUCHING;
}

fFDer0 = PseudodistanceDerivative(fT0,fF0,
    rkVelocity0,rkVelocity1);
if (fFDer0 >= 0.0f)
{
    // The objects are moving apart.
    rfContactTime = -Mathf::MAX_REAL;
    return SEPARATED;
}
}

// Newton's method failed to converge, but we already
// tested earlier if the objects were moving apart or
// not intersecting during the specified time interval.
// To reach here, the number of iterations was not large
// enough for the desired pseudodistance threshold. Most
// likely, this occurs when the relative speed is very
// large and the time step for the derivative estimation
// needs to be smaller.
rfContactTime = fT0;
ComputeContactInformation(TOUCHING,rfContactTime,

```

```

        rkVelocity0,rkVelocity1);
    return TOUCHING;
}

```

The base class implementation for estimating the pseudodistance derivative is

```

float Colliders::PseudodistanceDerivative (float fT0,
    float fF0, const Vector3f& rkVelocity0,
    const Vector3f& rkVelocity1) const
{
    float fT1 = fT0 - m_fDerivativeTimeStep;
    float fF1 = Pseudodistance(fT1,rkVelocity0,rkVelocity1);
    float fFDer0 = (fF0 - fF1) * m_fInvDerivativeTimeStep;
    return fFDer0;
}

```

This code uses a backward difference for the estimation.

8.3.2 PSEUDODISTANCES FOR SPECIFIC PAIRS OF OBJECT TYPES

A compendium of pseudodistance functions is presented here for use in classes derived from Colliders.

Sphere-Swept Volumes

The simplest object types to work with are spheres. If the spheres have centers $\mathbf{C}_i$ and radii r_i for $i = 0, 1$, they are separated when $|\mathbf{C}_1 - \mathbf{C}_0| > r_0 + r_1$, just touching when $|\mathbf{C}_1 - \mathbf{C}_0| = r_0 + r_1$, and overlapping when $|\mathbf{C}_1 - \mathbf{C}_0| < r_0 + r_1$. This suggests using the pseudodistance function

$$|\mathbf{C}_1 - \mathbf{C}_0| - (r_0 + r_1)$$

However, we can avoid the square root calculation in computing the length of the difference of centers, and we can provide some normalization based on size. This normalization is important in choosing the threshold for comparing $G(t)$ to zero. My preference is to use

```

float Pseudodistance (Sphere sphere0, Sphere sphere1)
{
    Vector3 diff = sphere1.C - sphere0.C;
    float sd = diff.SquaredLength();
    float rsum = sphere0.r + sphere1.r;
    return sd/(rsum*rsum) - 1;
}

```


Separation or overlap is a simple matter for a sphere and a capsule. The distance between the sphere center and capsule segment is compared to the sum of the radii of the objects. The three cases are similar to what we saw for the sphere-sphere case. A pseudodistance function is

```
float Pseudodistance (Sphere sphere, Capsule capsule)
{
    float sd = SquaredDistance(sphere.C, capsule.segment);
    float rsum = sphere.r + capsule.r;
    return sd/(rsum*rsum) - 1;
}
```

The point-segment, squared-distance function is discussed in Section 14.1.3.

A sphere and lozenge are handled similarly to a sphere and capsule. The distance between the sphere center and lozenge rectangle is compared to the sum of the radii of the objects. The three cases are similar to what we saw for the sphere-sphere case. A pseudodistance function is

```
float Pseudodistance (Sphere sphere, Lozenge lozenge)
{
    float sd = SquaredDistance(sphere.C, lozenge.rectangle);
    float rsum = sphere.r + lozenge.r;
    return sd/(rsum*rsum) - 1;
}
```

The point-rectangle, squared-distance function is discussed in Section 14.5.

The distance between the capsule segments is compared to the sum of the radii of the objects. A pseudodistance function is

```
float Pseudodistance (Capsule capsule0, Capsule capsule1)
{
    float sd = SquaredDistance(capsule0.segment,
                               capsule1.segment);
    float rsum = capsule0.r + capsule1.r;
    return sd/(rsum*rsum) - 1;
}
```

The segment-segment, squared-distance function is discussed in Section 14.2.6.

The distance between a capsule segment and a lozenge rectangle is compared to the sum of the radii of the objects. A pseudodistance function is

```
float Pseudodistance (Capsule capsule, Lozenge lozenge)
{
    float sd = SquaredDistance(capsule.segment,
```

```

    lozenge.rectangle);
float rsum = capsule.r + lozenge.r;
return sd/(rsum*rsum) - 1;
}

```

The segment-rectangle, squared-distance function is discussed in Section 14.6.3.

Finally, two lozenges are handled by the following pseudodistance function. The distance between the lozenge rectangles is compared to the sum of the radii of the objects.

```

float Pseudodistance (Lozenge lozenge0, Lozenge lozenge1)
{
    float sd = SquaredDistance(lozenge0.rectangle,
                                lozenge1.rectangle);
    float rsum = lozenge0.r + lozenge1.r;
    return sd/(rsum*rsum) - 1;
}

```

The rectangle-rectangle, squared-distance function is discussed in Section 14.7.

As you can see from these simple examples, sphere-swept volumes nicely support the concept of pseudodistance.

Sphere and Box

The natural choice for pseudodistance between a box and a sphere is to compute the distance from the sphere center to the box and subtract the radius. This quantity is positive when the objects are separated, zero if they are just touching, and negative when they are overlapping. To avoid the square root calculation in the distance from point to box, squared distance is used instead. So we can look at the squared distance from box to sphere center minus the squared sphere radius. This measurement is an absolute one in the sense that it varies with the scale of the objects. To obtain scale independence, a relative measurement should be used. The pseudodistance function for a box and capsule is

```

float Pseudodistance (Sphere sphere, Box box)
{
    float sd = SquaredDistance(sphere.C, box);
    return sd/(sphere.r*sphere.r) - 1;
}

```

The point-box, squared-distance function is discussed in Section 14.8.

Capsule and Box

The natural choice for pseudodistance between a box and a capsule is to compute the distance from the capsule segment to the box and subtract the radius. This quantity is positive when the objects are separated, zero if they are just touching, and negative when they are overlapping. The pseudocode is

```
float Pseudodistance (Capsule capsule, Box box)
{
    float sd = SquaredDistance(capsule.segment, box);
    return sd/(capsule.r*capsule.r) - 1;
}
```

The segment-box, squared-distance function is discussed in Section 14.9.3.

Sphere and Triangle

The natural choice for pseudodistance between a sphere and a triangle is to compute the distance from the sphere center to the triangle and subtract the radius. This quantity is positive when the objects are separated, zero if they are just touching, and negative when they are overlapping.

```
float Pseudodistance (Sphere sphere, Triangle triangle)
{
    float sd = SquaredDistance(sphere.C, triangle);
    return sd/(sphere.r*sphere.r) - 1;
}
```

The point-triangle, squared-distance function is discussed in Section 14.3.

Capsule and Triangle

The natural choice for pseudodistance between a capsule and a triangle is to compute the distance from the capsule segment to the triangle and subtract the radius. This quantity is positive when the objects are separated, zero if they are just touching, and negative when they are overlapping.

```
float Pseudodistance (Capsule capsule, Triangle triangle)
{
    float sd = SquaredDistance(capsule.segment, triangle);
    return sd/(capsule.r*capsule.r) - 1;
}
```

The segment-triangle, squared-distance function is discussed in Section 14.4.3.

Convex Polygons and Convex Polyhedra

We have already seen that the method of separating axes may be used to compute the first time of contact between convex polygons or convex polyhedra moving with constant linear velocities. Alternatively, we may use the collision detection system described here using pseudodistances between *stationary* polygons or polyhedra. Recall that two objects are separated if the projections of the objects onto one of N different axes are separated. If all the projections on the N axes overlap, then the objects overlap. The number N depends on whether the objects are polygons or polyhedra and depends on the number of edges of the polygons and the number of faces of the polyhedra.

We may use the computations from the separation tests to build a pseudodistance function. When all axes report overlap, choose the pseudodistance to be the negative of the length of the *largest* overlapping interval (selected from N overlapping intervals). When one axis reports no overlap, the objects are separated. A test-intersection query is happy with this information and reports immediately that there is separation. The find-intersection query using pseudodistance needs to do more work. The projections are still processed for N axes. The pseudodistance is chosen to be the (positive) length of the *smallest* overlapping interval. If you do not process all axes, there is the potential for the pseudodistance to be a discontinuous function, which might affect the Newton's iterations. By choosing the smallest length, you necessarily will trap the case of just-touching objects—the smallest length is zero because the projection intervals are just touching.

8.3.3 COLLISION CULLING WITH AXIS-ALIGNED BOUNDING BOXES

Detecting contact between nearby objects is sometimes referred to as the *narrow phase* of collision detection. The *broad phase* is designed to eliminate pairs of objects that are simply not in the proximity of each other. A particularly effective system uses axis-aligned bounding boxes (AABBs) for the convex polyhedra bounding volumes. If the AABBs of the objects do not intersect, then there is no need to perform the more expensive intersection query for the convex polyhedra bounding volumes. The collision culling system described here uses both spatial and temporal coherence.

Each time step that the convex polyhedra move, their AABBs move. First, we need to update the AABB to make sure it contains the polyhedron. The polyhedron could have rotated in addition to being translated, so it is not enough just to translate the AABB. An iteration over the vertices of the newly moved polyhedron may be used to compute the extremes along each coordinate axis, but if we have added the fast extremal query support to the convex polyhedra discussed earlier, we can use six queries to compute the AABB. It is also possible to use an AABB that is guaranteed to contain any rotated version of the polyhedron, in which case we need only translate the AABB when the polyhedron translates and/or rotates. This option avoids all the extremal queries in exchange for a looser-fitting bounding box.

Second, once the AABBs are updated for all the polyhedra, we expect that the intersection status of pairs of polyhedra/AABBs has changed—old intersections might no longer exist, new intersections might now occur. Spatial and temporal coherence will be used to make sure the update of status is efficient.

Intersecting Intervals

The idea of determining intersection between AABBs is based on sorting and update of intervals on the real line, a 1D problem that we will analyze first. The method we describe here is mentioned in [Bar01]. A more general discussion of intersections of rectangles in any dimension is provided in [PS85]. Consider a collection of n intervals $I_i = [b_i, e_i]$ for $1 \leq i \leq n$. The problem is to efficiently determine all pairs of intersecting intervals. The condition for a single pair I_i and I_j to intersect is $b_j \leq e_i$ and $b_i \leq e_j$. The naive algorithm for the full set of intervals just compares all possible pairs, an $O(n^2)$ algorithm.

A more efficient approach uses a *sweep algorithm*, a concept that has been used successfully in many computational geometry algorithms. First, the interval endpoints are sorted into ascending order. An iteration is made over the sorted list (the sweep) and a set of *active intervals* is maintained, initially empty. When a beginning value b_i is encountered, all active intervals are reported as intersecting with interval I_i , and I_i is added to the set of active intervals. When an ending value e_i is encountered, interval I_i is removed from the set of active intervals. The sorting phase is $O(n \log n)$. The sweep phase is $O(n)$ to iterate over the sorted list, clearly asymptotically faster than $O(n \log n)$. The intersecting reporting phase is $O(m)$ to report the m intersecting intervals. The total order is written as $O(n \log n + m)$. The worst-case behavior is when all intervals overlap, in which case $m = O(n^2)$, but for our applications we expect m to be relatively small. Figure 8.24 illustrates the sweep phase of the algorithm.

The sorted interval endpoints are shown on the horizontal axis of the figure. The set of active intervals is initially empty, $A = \emptyset$. The first five sweep steps are enumerated as follows:

1. b_3 encountered. No intersections reported since A is empty. Update $A = \{I_3\}$.
2. b_1 encountered. Intersection $I_3 \cap I_1$ is reported. Update $A = \{I_3, I_1\}$.
3. b_2 encountered. Intersections $I_3 \cap I_2$ and $I_1 \cap I_2$ reported. Update $A = \{I_3, I_1, I_2\}$.
4. e_3 encountered. Update $A = \{I_1, I_2\}$.
5. e_1 encountered. Update $A = \{I_2\}$.

The remaining steps are easily stated and are left as an exercise.

A warning is in order here: The sorting of the interval endpoints must be handled carefully when equality occurs. For example, suppose that two intervals $[b_i, e_i]$ and $[b_j, e_j]$ intersect in a single point, $e_i = b_j$. If the sorting algorithm lists e_i before

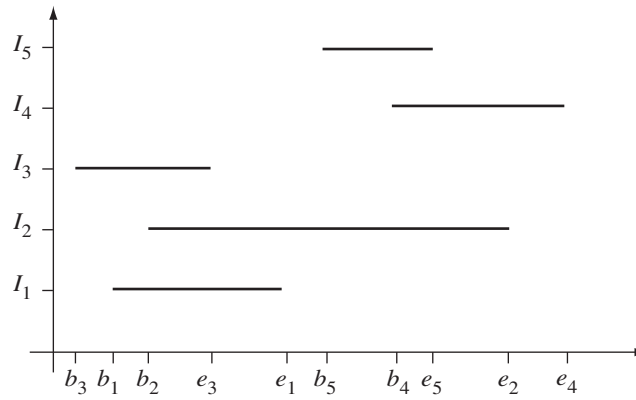


Figure 8.24 The sweep phase of the algorithm.

b_j , then when e_i is encountered in the sweep, we remove I_i from the set of active intervals. Next, b_j is encountered and intersections of I_j with the active intervals are reported. The interval I_i was removed from the active set on the previous step, so $I_j \cap I_i$ is *not reported*. In the sort, suppose instead that b_j is listed before e_i by the sorting algorithm. Since b_i was encountered earlier in the sweep, the set of active intervals contains I_i . When b_j is encountered, $I_j \cap I_i$ is *reported* as an intersection. Clearly, the order of equal values in the sort is important. Our application will require that we report just-touching intersections, so the interval endpoints cannot be sorted just as a set of floating-point numbers. Tags need to be associated with each endpoint indicating whether it is a beginning point or an ending point. The sorting must take the tag into account to make sure that equal endpoint values are sorted, so that values with a “begin” tag occur before values with an “end” tag. The tags are not a burden since, in fact, we need them anyway to decide during the sweep what type of endpoint we have encountered. Pseudocode for the sort and sweep is

```
struct EndPoint
{
    enum Type { BEGIN = 0, END = 1 };
    Type type;
    double value;
    int interval; // index of interval containing endpoint

    // EndPoint E1, E2;
    // E1 < E2 when
    //   E1.value < E2.value, or
```

```

    // E1.value == E2.value AND E1.type < E2.type
}

struct Interval
{
    EndPoint P[2];
};

void SortAndSweep (int n, Interval I[])
{
    // Use  $O(n \log n)$  sort.
    array<EndPoint> L = Sort(n,I);

    // active set of intervals (stored by index in array)
    set<int> A = empty;

    // (i,j) in S means I[i] and I[j] overlap.
    set<int,int> S = empty;

    for (i = 0; i < L.size(); i++)
    {
        if ( L[i].type == EndPoint::BEGIN )
        {
            for (each j in A) do
                S.Insert(j,L[i].interval);
            A.Insert(L[i].interval);
        }
        else // L[i].type == EndPoint::END
        {
            A.Remove(I[L[i].interval]);
        }
    }
}

```

Once the sort and sweep has occurred, the intervals are allowed to move about, thus invalidating the order of the endpoints in the sorted list. We can re-sort the values and apply another sweep, an $O(n \log n + m)$ process. However, we can do better than that. The sort itself may be viewed as a way to know the spatial coherence of the intervals. If the intervals move only a small distance, we expect that not many of the endpoints will swap order with their neighbors. The modified list is *nearly sorted*, so we should re-sort using an algorithm that is fast for nearly sorted inputs. Taking advantage of the small number of swaps is our way of using temporal coherence to reduce our workload. The insertion sort is a fast algorithm for sorting nearly sorted

lists. For general input it is $O(n^2)$, but for nearly sorted data it is $O(n + e)$, where e is the number of exchanges used by the algorithm. Pseudocode for the insertion sort is

```
// input: A[0] through A[n-1]
// output: array sorted in-place
void InsertionSort (int n, type A[])
{
    for (j = 1; j < n; j++)
    {
        key = A[j];
        i = j - 1;
        while ( i >= 0 and A[i] > key )
        {
            Swap(A[i],A[i+1]);
            i--;
        }
        A[i+1] = key;
    }
}
```

The situation so far is that we have applied the sort-and-sweep algorithm to our collection of intervals, a once-only step that requires $O(n \log n + m)$ time. The output is a set S of pairs (i, j) that correspond to overlapping intervals, $I_i \cap I_j$. Some intervals are now moved, and the list of endpoints is re-sorted in $O(n + e)$ time. The set S might have changed. Two overlapping intervals might not overlap now. Two nonoverlapping intervals might now overlap. To update S we can simply apply the sweep algorithm from scratch, an $O(n + m)$ algorithm, and build S anew. Better, though, is to mix the update with the insertion sort. An exchange of two “begin” points with two “end” points does not change the intersection status of the intervals. If a pair of “begin” and “end” points is swapped, then we have either gained a pair of overlapping intervals or lost a pair. By temporal coherence, we expect the number of changes in status to be small. If c is the number of changes of overlapping status, we know that $c \leq e$, where e is the number of exchanges in the insertion sort. The value e is expected to be much smaller than m , the number of currently overlapping intervals. Thus, we would like to avoid the full sweep that takes $O(n + m)$ time and update during the insertion sort that takes shorter time $O(n + e)$.

Figure 8.25 illustrates the update phase of the algorithm applied to the intervals shown in Figure 8.24. At the initial time, the sorted endpoints are $\{b_3, b_1, b_2, e_3, e_1, b_5, b_4, e_5, e_2, e_4\}$. The pairs of indices for the overlapping intervals are $S = \{(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (4, 5)\}$. Now I_1 moves to the right and I_5 moves to the left. The new endpoints are denoted $\bar{b}_1, \bar{e}_1, \bar{b}_5$, and $\bar{e}_5$. The list of endpoints that was

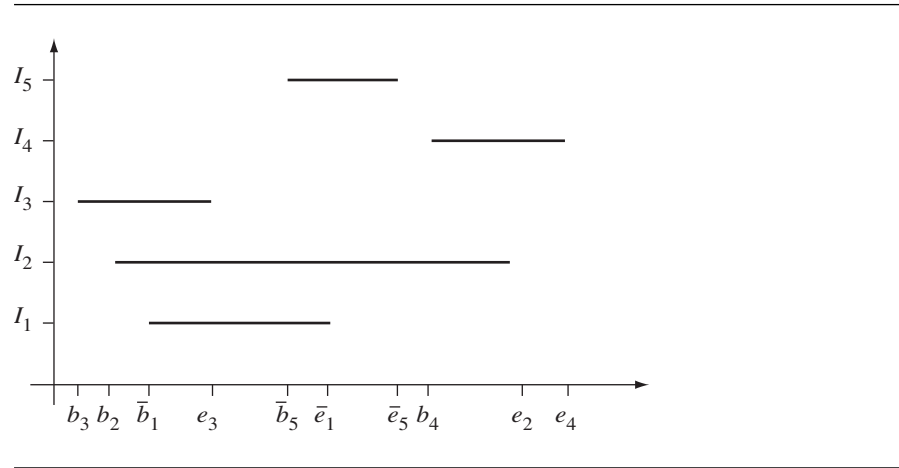


Figure 8.25 The update phase of the algorithm when intervals have moved.

sorted but now has had values changed is $\{b_3, \bar{b}_1, b_2, e_3, \bar{e}_1, \bar{b}_5, b_4, e_5, e_2, e_4\}$. The insertion sort is applied to this set of values. The steps follow.

1. Initialize the sorted list to be $L = \{b_3\}$.
2. Insert $\bar{b}_1$, $L = \{b_3, \bar{b}_1\}$.
3. Insert b_2 , $L = \{b_3, \bar{b}_1, b_2\}$.
 - (a) Exchange $\bar{b}_1$ and b_2 , $L = \{b_3, b_2, \bar{b}_1\}$. No change to S .
4. Insert e_3 , $L = \{b_3, b_2, \bar{b}_1, e_3\}$.
5. Insert $\bar{e}_1$, $L = \{b_3, b_2, \bar{b}_1, e_3, \bar{e}_1\}$.
6. Insert $\bar{b}_5$, $L = \{b_3, b_2, \bar{b}_1, e_3, \bar{e}_1, \bar{b}_5\}$.
 - (a) Exchange $\bar{e}_1$ and $\bar{b}_5$, $L = \{b_3, b_2, \bar{b}_1, e_3, \bar{b}_5, \bar{e}_1\}$. This exchange causes I_1 and I_5 to overlap, so insert $(1, 5)$ into the set: $S = \{(1, 2), (1, 3), (1, 5), (2, 3), (2, 4), (2, 5), (4, 5)\}$.
7. Insert b_4 , $L = \{b_3, b_2, \bar{b}_1, e_3, \bar{b}_5, \bar{e}_1, b_4\}$.
8. Insert $\bar{e}_5$, $L = \{b_3, b_2, \bar{b}_1, e_3, \bar{b}_5, \bar{e}_1, b_4, \bar{e}_5\}$.
 - (a) Exchange b_4 and $\bar{e}_5$, $L = \{b_3, b_2, \bar{b}_1, e_3, \bar{b}_5, \bar{e}_1, \bar{e}_5, b_4\}$. This exchange causes I_4 and I_5 to no longer overlap, so remove $(4, 5)$ from the set: $S = \{(1, 2), (1, 3), (1, 5), (2, 3), (2, 4), (2, 5)\}$.
9. Insert e_2 , $L = \{b_3, b_2, \bar{b}_1, e_3, \bar{b}_5, \bar{e}_1, \bar{e}_5, b_4, e_2\}$.
10. Insert e_4 , $L = \{b_3, b_2, \bar{b}_1, e_3, \bar{b}_5, \bar{e}_1, \bar{e}_5, b_4, e_2, e_4\}$.
11. The new list is sorted and the set of overlaps is current.

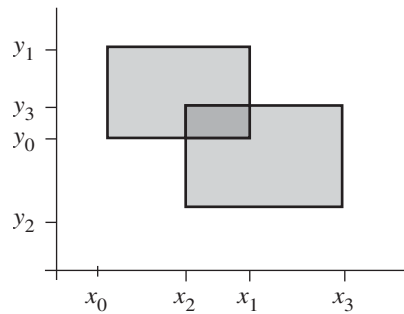


Figure 8.26 Axis-aligned rectangles overlap when both their x -intervals and y -intervals overlap.

Intersecting Rectangles

The algorithm for computing all pairs of intersecting axis-aligned rectangles is a simple extension of the algorithm for intervals. An axis-aligned rectangle is of the form $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$. Two such rectangles intersect if there is overlap between both their x -intervals and their y -intervals, as shown in Figure 8.26. The rectangles are $[x_0, x_1] \times [y_0, y_1]$ and $[x_2, x_3] \times [y_2, y_3]$. The rectangles overlap because $[x_0, x_1] \cap [x_2, x_3] \neq \emptyset$ and $[y_0, y_1] \cap [y_2, y_3] \neq \emptyset$.

In the 2D setting we maintain two sorted lists, one for the endpoints of the x -intervals and one for the endpoints of the y -intervals. The initial step of the algorithm sorts the two lists. The sweep portion is only slightly more complicated than for one dimension. The condition for overlap is that the x -intervals *and* y -intervals overlap. If we were to sweep the sorted x -list first and determine that two x -intervals overlap, that is not sufficient to say that the rectangles of those x -intervals overlap. We could devise some fancy scheme to sweep both x - and y -lists at the same time, but it is simpler just to do a little extra work. If two x -intervals overlap, we will test for overlap of the corresponding rectangles in both dimensions and update the set of overlapping rectangles as needed.

Once we have the sorted lists and a set of overlapping rectangles, we will move the rectangles and must update the lists and set. The process will use an insertion sort to take advantage of spatial and temporal coherence. The x -list is processed first. If an exchange occurs so that two previously overlapping intervals no longer overlap, the corresponding rectangles no longer overlap so we can remove that pair from the set of overlaps. If an exchange occurs so that two previously nonoverlapping intervals now overlap, the corresponding rectangles may or may not overlap. Just as we did for the initialization phase, we will simply test the corresponding rectangles for overlap in both dimensions and adjust the set of overlaps accordingly.

Intersecting Boxes

You should see clearly that the algorithm for axis-aligned rectangles in two dimensions extends easily to axis-aligned boxes in three dimensions. The collision system itself has the following outline:

1. Generate AABBs for the convex polyhedra of the system using the fast extremal query support built into the polyhedra.
2. Using the sort-and-sweep method, compute the set S of all pairs of intersecting AABBs.
3. Determine which AABBs intersect using the fast insertion sort update.
4. For each pair of intersecting AABBs, determine if the contained convex polyhedra intersect. For those pairs that do, compute the contact information.
5. The contact information is passed to the collision response system to be used to modify the behavior of the objects in a physically realistic manner. Typically, this amounts to changing the path of motion, adjusting the linear and angular velocities, and transferring momentum and energy, and is done via differential equation solvers for the equations of motion.
6. Recompute the AABBs using the fast extremal query support.
7. Repeat step 3.

8.4 OBJECT PICKING

A classical application for line-object intersection is *picking*—selecting an object drawn on the screen by clicking a pixel in that object using the mouse. Figure 8.27 illustrates a *pick ray* associated with a screen pixel, where the screen is associated with the near plane of a view frustum. The eye point $\mathbf{E}$, in world coordinates, is used for the origin of the pick ray. We need to compute a unit-length direction $\mathbf{W}$, in world coordinates. The pick ray is then $\mathbf{E} + t\mathbf{W}$ for $t \geq 0$.

8.4.1 CONSTRUCTING A PICK RAY

The screen has a width of W pixels and a height of H pixels, and the screen coordinates are left-handed (x to the right, y down). The selected point (x, y) is in screen coordinates, where $0 \leq x \leq W - 1$ and $0 \leq y \leq H - 1$. This point must be converted to world coordinates for points on the near plane. Specifically, we need a corresponding point $\mathbf{Q} = \mathbf{E} + n\mathbf{D} + x_v\mathbf{R} + y_v\mathbf{U}$, where $r_{\min} \leq x_v \leq r_{\max}$ and $u_{\min} \leq y_v \leq u_{\max}$, and where n is the distance from the eye point to the near plane. This is a matter of some simple algebra:

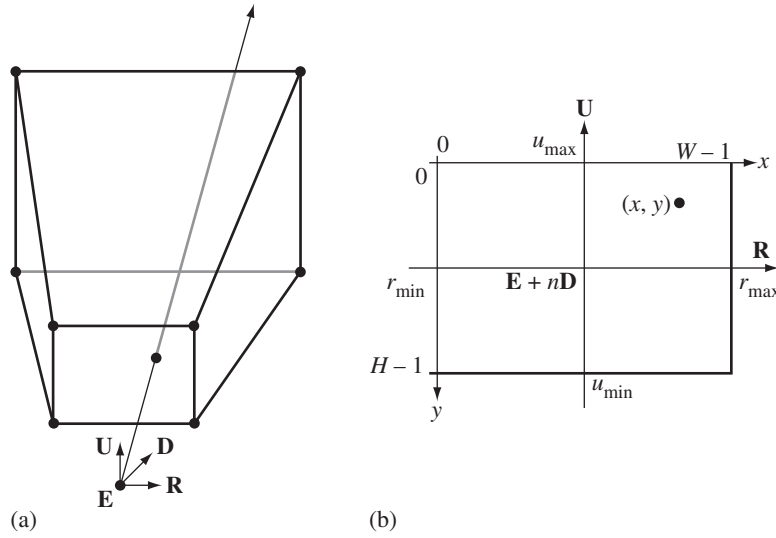


Figure 8.27 (a) A view frustum with a point selected on the near plane. The pick ray has an origin that is the eye point E , in world coordinates, and a direction that is from the eye point to the selected point. (b) The viewport on the near plane with screen coordinates (x, y) listed for the selected point.

$$(x_p, y_p) = \left(\frac{x}{W-1}, 1 - \frac{y}{H-1} \right)$$

$$(x_v, y_v) = ((1 - x_p)r_{\min} + x_p r_{\max}, (1 - y_p)u_{\min} + y_p u_{\max})$$

The first equation lists the *normalized viewport coordinates*, (x_p, y_p) , that live in the set $[0, 1]^2$. Observe that the left-handed screen coordinates are converted to right-handed normalized viewport coordinates by reflecting the y -value. The pick ray direction is

$$\mathbf{W} = \frac{\mathbf{Q} - \mathbf{E}}{|\mathbf{Q} - \mathbf{E}|} = \frac{n\mathbf{D} + x_v\mathbf{R} + y_v\mathbf{U}}{n^2 + x_v^2 + y_v^2}$$

The pick ray may now be used for intersection testing with objects in the world to identify which one has been selected.

The construction is accurate as long as the *entire viewport* is used for drawing the scene. The Camera class, however, allows you to select a subrectangle on the screen in which the scene is drawn, via member function `SetViewport`. Let x_{pmin} , x_{pmax} , y_{pmin} , and y_{pmax} be the viewport settings in the camera class. The default minimum values

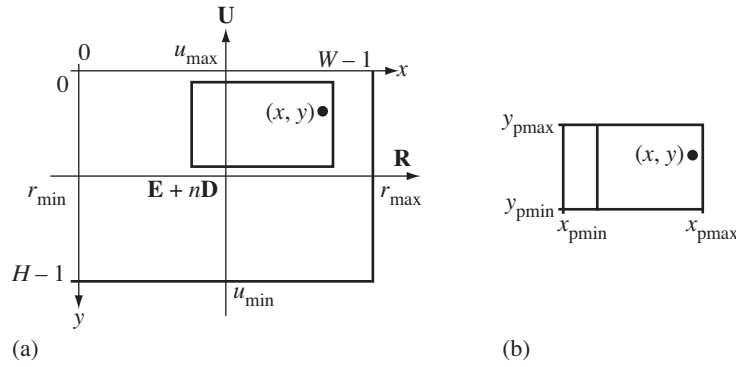


Figure 8.28 The viewport on the near plane with screen coordinates (x, y) listed for the selected point. The viewport is not the entire screen.

are 0, and the default maximum values are 1. If they are changed from the defaults, the pick ray construction must be modified. Figure 8.28 shows the screen with a viewport that is not the size of the screen.

The new construction for (x_v, y_v) is

$$\begin{aligned} (x_p, y_p) &= \left(\frac{x}{W-1}, 1 - \frac{y}{H-1} \right) \\ (x_w, y_w) &= \left(\frac{x_p - x_{pmin}}{x_{pmax} - x_{pmin}}, \frac{y_p - y_{pmin}}{y_{pmax} - y_{pmin}} \right) \\ (x_v, y_v) &= ((1 - x_w)r_{min} + x_w r_{max}, (1 - y_w)u_{min} + y_w u_{max}) \end{aligned}$$

The conversion is as if the smaller viewport really did fill the entire screen. This makes sense in that the world coordinate pick ray should be the same for a scene regardless of whether you draw the scene in the full screen or in a subrectangle of the screen. The implementation for constructing the pick ray is the function `Camera::GetPickRay`, found in `Wm4Camera.cpp` on the CD-ROM. The `Camera` class does not store the width and height of the screen, so those values must be passed to the function. The reason not to store them is that the screen dimensions are allowed to change, for example, via resizing of the application window, but the camera model is not dependent on those changes. The `Renderer` class does store the screen dimensions.

8.4.2 SCENE GRAPH SUPPORT

Now that we know how to construct a pick ray, we actually have to do the intersection testing with the scene. Wild Magic supports this in a hierarchical manner, using the bounding volumes attached to the nodes in a scene hierarchy. Starting with the root node, a test-intersection query is made between the pick ray and the world bounding volume of the node. If the ray does not intersect the bounding volume, then it cannot intersect the scene contained in the bounding volume—no intersection occurs. If the pick ray does intersect the world bounding volume, then the test-intersection query is propagated to all the children of the node (in the usual depth-first manner). The propagation continues recursively along a path to a leaf node as long as the ray and bounding volumes along that path intersect. If the ray and leaf-node bounding volume intersect, then a find-intersection query between the ray and whatever the leaf node represents must be made. The query depends on the actual class type of the leaf; the information reported by the query is also dependent on the class type.

The subsystem for hierarchical picking is contained in class `Spatial`. The relevant interface is

```
class Spatial
{
public:
    class PickRecord
    {
    public:
        virtual ~PickRecord ();

        Pointer<Spatial> IOobject; // intersected leaf object
        float T; // pick ray parameter (t >= 0)

    protected:
        PickRecord (Spatial* pkIOobject, float fT);
    };

    typedef TArray<PickRecord*> PickArray;

    virtual void DoPick (const Vector3f& rkOrigin,
        const Vector3f& rkDirection, PickArray& rkResults);

    static PickRecord* GetClosest (PickArray& rkResults);
};
```

The nested class `PickRecord` represents the minimal amount of information that a find-intersection query computes, the parameter t for which the pick ray $\mathbf{P} + t\mathbf{D}$ intersects an object. Each `Spatial`-derived class derives its own pick record class from

PickRecord and adds whatever information it wants to return from a successful find-intersection query, instigated by a call to DoPick. The ray parameter can be used to sort the intersection points after a call to DoPick.

Notice that the PickRecord does not have the capability for run-time type information (RTTI). However, RTTI is obtained by using the Object-based RTTI for the PickRecord data member IObject. Once that member's type is known, the PickRecord can be cast to the appropriate PickRecord-derived class.

A find-intersection query can produce a lot of intersection results. Thus, a container class for pick records is needed to store the results. I have chosen a dynamic array, named PickArray, that stores an array of pointers to PickRecord objects. The entry point to the query is the function DoPick. The inputs are the origin and direction for the pick ray, in world coordinates, and an array to store the pick records. On return, the caller is responsible for iterating over the array and deleting all the PickRecord objects.

In most cases, the picked object is the one closest to the origin of the ray. The function GetClosest implements a simple search of the array for the pick record with the minimum t -value.

The Node class is responsible for the test-intersection query between the pick ray and world bounding volume and for propagating the call to its children if necessary. The DoPick function is

```
void Node::DoPick (const Vector3f& rkOrigin,
                  const Vector3f& rkDirection, PickArray& rkResults)
{
    if (WorldBound->TestIntersection(rkOrigin,rkDirection))
    {
        for (int i = 0; i < m_kChild.GetQuantity(); i++)
        {
            Spatial* pkChild = m_kChild[i];
            if (pkChild)
            {
                pkChild->DoPick(rkOrigin,rkDirection,
                               rkResults);
            }
        }
    }
}
```

The implementation is straightforward. The BoundingBox class has support for the test-intersection query with a ray. If the ray intersects the bounding volume, an iteration is made over the children and the call is propagated. The SwitchNode class has a similar implementation, but it only propagates the call to the active child.

The most relevant behavior of DoPick is in the class TriMesh, which represents a mesh of triangles. The class introduces its own PickRecord type:

```

class TriMesh
{
public:
    class PickRecord : public Geometry::PickRecord
    {
    public:
        PickRecord (TriMesh* pkIObject, float fT,
                    int iTriangle, float fBary0, float fBary1,
                    float fBary2);

        // index of the triangle intersected by the ray
        int Triangle;

        // barycentric coordinates of point of intersection
        // If b0, b1, and b2 are the values, then all are
        // in [0,1] and b0 + b1 + b2 = 1.
        float Bary0, Bary1, Bary2;
    };
};

```

The pick record stores the index of any triangle intersected by the ray. It stores the barycentric coordinates of the intersection point with respect to the triangle. This allows the application to compute interpolated vertex attributes as well as the actual point of intersection. The base class for the pick record is `Geometry::PickRecord`, but in fact `Geometry` has no such definition. The compiler resorts to checking farther down the line and finds `Spatial::PickRecord` as the base class.

The implementation of `DoPick` is

```

void TriMesh::DoPick (const Vector3f& rkOrigin,
                     const Vector3f& rkDirection, PickArray& rkResults)
{
    if (WorldBound->TestIntersection(rkOrigin,rkDirection))
    {
        // Convert the ray to model-space coordinates.
        Ray3f kRay(World.ApplyInverse(rkOrigin),
                   World.InvertVector(rkDirection));

        // Compute intersections with the model-space triangles.
        const Vector3f* akVertex = Vertices->GetData();
        int iTQuantity = Indices->GetQuantity()/3;
        const int* piConnect = Indices->GetData();
        for (int i = 0; i < iTQuantity; i++)
        {
            int iV0 = *piConnect++;

```



```

        int iV1 = *piConnect++;
        int iV2 = *piConnect++;

        Triangle3f kTriangle(akVertex[iV0],akVertex[iV1],
                               akVertex[iV2]);
        IntrRay3Triangle3f kIntr(kRay,kTriangle);
        if (kIntr.Find())
        {
            rkResults.Append(new PickRecord(this,
                                              kIntr.GetRayT(),i,kIntr.GetTriB0(),
                                              kIntr.GetTriB1(),kIntr.GetTriB2()));
        }
    }
}

```

A test-intersection query is made between the pick ray and the world bounding volume. If the ray intersects the bounding volume, a switch is made to a find-intersection query to determine which triangles are intersected by the ray, and where. The triangle vertex data is in model coordinates, whereas the pick ray is in world coordinates. We certainly could transform each model triangle to world space and call the find-intersection query, but that involves potentially a large number of vertex transformations—an expense you do not want to incur. Instead, the pick ray is inverse-transformed to the model space of the triangle mesh, and the find-intersection queries are executed. This is a much cheaper alternative! For each triangle, a ray-triangle, find-intersection query is made. If an intersection occurs, a pick record is created and inserted into the array of pick records.

One potential inefficiency is that the triangles are processed in a linear traversal. If the mesh has a very large number of triangles and the ray intersects only a very small number, a lot of computational time will be spent finding out that many triangles are not intersected by the ray. This is a fundamental problem in ray-tracing applications. One of the solutions is to use a hierarchical bounding volume tree that fits the triangle mesh. The idea is to localize in the mesh where intersections might occur by culling out large portions of the bounding volume tree, using fast rejection algorithms for ray-object pairs. Well, this is exactly what we have done at a coarse level, where the nodes of the tree are the Node objects in the scene. I have not provided a fine-level decomposition at the triangle mesh level *for the purposes of picking*, but it certainly can be added if needed. For example, you could implement an OBB tree to help localize the calculations [GLM96].

A couple of the sample applications use picking. The application

allows you to pick objects in the scene. If you pick using the left mouse button, the name of the selected geometry object is displayed in the lower-right portion of the screen. If you pick using the right mouse button, the selected geometry object is displayed as a wireframe object. This allows you to see what is behind the scenes (pun intended).

Another application using picking is

`GeometricTools/WildMagic4/SampleGraphics/MorphControllers`

This application displays a morphing face in the lower portion of the screen. A reduced viewport is used for the display. The upper portion of the screen displays the five targets of the morph controller. Each target is displayed in its own small viewport. The picking system reports which target you have selected, or if you selected the morphing face, or if nothing has been selected. The application illustrates that the pick ray must be chosen using the camera's viewport settings and cannot always assume the viewport is the entire screen.

8.4.3 STAYING ON TOP OF THINGS

Given a 3D environment in which characters can roam around, an important feature is to make certain that the characters stay on the ground and not fall through! If the ground is a single plane, you may use this knowledge to keep the camera (the character's eye point, so to speak) at a fixed height about the plane. However, if the ground is terrain, or a set of steps or floors in an indoor level, or a ramp, or any other nonplanar geometry, the manual management of the camera's height above the current ground location becomes more burdensome than you might like.

The picking system can help you by eliminating a lot of the management. The idea is to call the picking system each time the camera moves. The pick ray has origin E (the camera eye point) and direction $-U$ (the downward direction) for the environment. Do not use the camera's up vector for U . If you were to pitch forward to look at the ground, the up vector rotates with you. The environment up vector is always fixed. The smallest t -value from the picking tells you the distance from the eye point to the closest object below you. You can then update the height of the camera using the t -value to make certain you stay at a fixed height.

The application

`GeometricTools/WildMagic4/SampleGraphics/Castle`

implements such a system. The relevant application function is

```
void Castle::AdjustVerticalDistance ()
{
    // Retain vertical distance above "ground."
```

```

    Spatial::PickArray kResults;
    m_spkScene->DoPick(m_spkCamera->GetLocation(),
        Vector3f(0.0f,0.0f,-1.0f), kResults);

    if (kResults.GetQuantity() > 0)
    {
        Spatial::PickRecord* pkRecord =
            Spatial::GetClosest(kResults);
        assert( pkRecord );
        TriMesh* pkMesh = DynamicCast<TriMesh>(
            pkRecord->IObject);
        TriMesh::PickRecord* pkTMRecord =
            (TriMesh::PickRecord*)pkRecord;

        Vector3f kV0, kV1, kV2;
        pkMesh->GetWorldTriangle(pkTMRecord->Triangle,
            kV0,kV1,kV2);
        Vector3f kClosest =
            pkTMRecord->Bary0*kV0 +
            pkTMRecord->Bary1*kV1 +
            pkTMRecord->Bary2*kV2;

        kClosest.Z() += m_fVerticalDistance;
        m_spkCamera->SetLocation(kClosest);

        for (int i = 0; i < kResults.GetQuantity(); i++)
        {
            delete kResults[i];
        }
    }
}

void Castle::MoveForward ()
{
    Application::MoveForward();
    AdjustVerticalDistance();
}

```

The `MoveForward` function is called when the up arrow is pressed. The base class `MoveForward` translates the camera's eye point by a small amount in the direction of view. The `AdjustVerticalDistance` adjusts that translation in the environment up direction to maintain a constant height above the ground or other objects. As you wander around the castle environment, you will notice that you always stay on top of things, including the ground, stairs, and ramps.

8.4.4 STAYING OUT OF THINGS

We have seen how to stay on top of things by using the picking system. In the same 3D environment, it is also important not to let the characters walk through walls or other objects. *Collision avoidance*, as it is called, is a broad topic. Typically, the avoidance is based on a test-intersection query of the bounding volume of the character against the various objects in the environment. This is complicated to get right, and potentially expensive if the object-object intersection algorithm is complicated.

An alternative that is not exact, but just a heuristic, is to cast a small number of pick rays from the eye point out into the scene. Consider this a form of *probing* (sometimes called *stabbing*, in the occlusion culling literature) to see what objects might be in the way. The more dense the set of rays, the less likely you will accidentally allow the character to pass through an object. However, the more dense the set, the more expensive the picking computations become. The balance between number of pick rays and speed of the testing will, of course, depend on your environments.

The application

`GeometricTools/WildMagic4/SampleGraphics/Castle`

supports collision avoidance via picking in addition to maintaining a constant height above the ground.

8.5 PATHFINDING TO AVOID COLLISIONS

Pathfinding is the process of determining a navigable path from a *source location* to a *destination location*. Various obstacles may occur along the way that must be avoided. General 3D pathfinding is difficult to implement and is CPU-intensive. A reduction in dimension is useful to improve performance by eliminating some of the computation burden. A pathfinding system may be built essentially for 2D (horizontal) motion, but allows movement vertically when necessary and when allowed. Much of the data needed to support pathfinding is constructed off-line, again reducing the computational burden during run time of the application. However, the off-line construction requires extracting information from the scene to allow the construction of data structures for the pathfinding. The simplest way to put this information into the 3D environment in the first place is to have an artist tag geometric quantities of interest in a scene build using his favorite 3D modeling package, export the scene to your own scene graph data structures, and then apply a postprocessing tool to generate the pathfinding data structures from the tags.

The essence of pathfinding is *visibility* and *reachability*. The destination is said to be *reachable* from the source if there is an unobstructed path connecting the two points. Such a path might exist even if the destination is not visible from the source. However, the path connecting the two points will consist of subpaths, with the endpoints of each subpath visible to each other. The essence of path construction will,

therefore, involve *visibility graphs*, a standard concept in computational geometry. Much of the material here should remind you of the discussion for a room-portal system since it also has a lot to do with visibility graphs.

8.5.1 ENVIRONMENTS, LEVELS, AND ROOMS

A scene graph is built to represent the 3D *environment*, which consists of a collection of *levels*. The typical scenario is an environment consisting of an outside region containing buildings, greenery, walkways, and other objects you normally find outside. The buildings consist of sets of *levels*, each level a collection of *rooms* interconnected by doorways, halls, or similar items. The levels themselves are interconnected by stairways, elevators, ramps, or similar items. The outside region is itself considered to be a level and may be thought of as partitioned into rooms, but only for the purposes of meeting some geometric constraints to make the pathfinding tractable. As a level, the outside is connected to buildings via doorways or other means of entering and exiting the building.

World Coordinate System

An environment has a world coordinate system consisting of an origin $\mathbf{C}$ and three direction vectors $\mathbf{D}$, $\mathbf{U}$, and $\mathbf{R}$; see Section 2.1.1. The up vector $\mathbf{U}$ provides the direction in which height is measured. The other two vectors are used for ground measurements. Any world point $\mathbf{X}$ may be represented in the world coordinate system as

$$\mathbf{X} = \mathbf{C} + g_0\mathbf{R} + g_1\mathbf{D} + h\mathbf{U}$$

The first two coefficients are the ground coordinates, (g_0, g_1) , and the last component is the height coordinate, h . All pathfinding computations are based on the world coordinate system and take advantage of the decomposition into ground coordinates and the height coordinate, a “2D plus 1D” decomposition.

Room-Doorway Multigraphs

The concepts of environment and level are fairly simple, both acting as grouping terms. An environment is a group of levels; a level is a group of rooms. The relationships between these entities are graph theoretic. The rooms in a level are connected by *doorways*, just like we had for a room-portal environment. At first glance you might be tempted to represent the rooms and doorways as an undirected graph, where the rooms are the graph nodes and the doorways are the graph arcs, but in fact the representation must be a *multigraph*. In a graph, two nodes may be connected by a single arc. In a level, two adjacent rooms may have multiple doorways between them; thus,

a graph is not sufficient for the representation and a multigraph is necessary. A room-portal system has the same requirement for visibility determination; see Section 6.3 for details.

Level-Connector Multigraphs

As mentioned previously, the levels are interconnected by stairways, elevators, ramps, or similar items. These objects connecting the levels are generally referred to as *connectors*, and the relationship between levels and connectors may be represented by a multigraph, just as rooms and doorways are. Multigraphs are necessary because two levels of a building may be connected by two or more stairways.

The Structure of Rooms

Each room represents a bounded region of space. When viewed from above, or equivalently when projected onto the ground plane, the room's projection is assumed to be bounded by a polygon. Many of the geometric queries needed to establish a navigable path between locations can be quite complicated if the bounding polygon is simple, but not convex. To simplify the implementation, maintain reasonable performance, and avoid the standard problems with floating-point, round-off errors in computational geometry, the assumption is made that the bounding polygon for the room's projection is convex. This does not preclude nonconvex regions that represent what you want to call a room, but like the room-portal system, you can partition the nonconvex region into convex ones, thus representing the true room as a union of convex rooms. The *boundary polygon* for each room may be constructed off-line by using the node tagging system that was described in Section 4.7.

The walls that define a room are assumed to be vertical. This assumption supports the construction of the boundary polygon for the room's projection. Although a reasonable assumption for many scenes, sometimes you might want walls that are not vertical or planar. This is possible through node tagging in the artwork. The “walls” used to define the room boundary need not be displayable geometry; that is, an artist can tag them to be removed after exporting or to be ignored by the renderer by setting the `Spatial::Culling` flags to `Spatial::CULL_ALWAYS`. The actual wall geometry must then be handled differently—as obstacles, a concept to be discussed later.

To complete the bounding geometry for a room, we also need a *floor* and a *ceiling*. These are triangle meshes that represent height fields. The meshes do not have to be coplanar triangles. This is particularly important for the outside region, which has no ceiling but has a floor that is most likely terrain. It is important, though, that these meshes represent height fields. This assumption, together with the convex boundary polygon assumption, supports fast point-in-room queries.

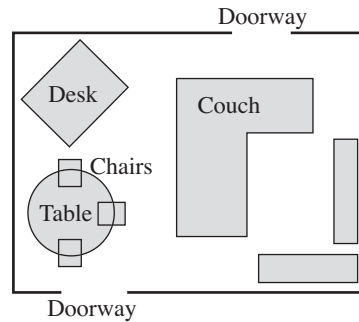


Figure 8.29 A typical blueprint of a room.

Each room has *obstacles* around which a game character must navigate. These are typically chairs, tables, desks, and so on. Given the fixed vertical direction, the room may be represented by a *blueprint*, which is the 2D layout of the room when viewed from the vertical direction. Figure 8.29 illustrates a typical blueprint. The doorways are cutouts of the walls and the obstacles are shaded in gray with black outlines. The arrangement of the table and chairs shows that the projections of the obstacles onto the horizontal are not necessarily separated. In Figure 8.29, the table is circular. Modeling packages produce triangle meshes, so in practice the table shape is polygonal.

The navigation problem is abstracted to finding a planar polyline path from a starting point to an ending point, where the polyline does not cut through polygonal obstacles. Generally, there are infinitely many paths, one of which is the shortest path. Figure 8.30 illustrates some paths. The taupe path is the shortest path connecting the starting and ending points. For polygonal obstacles, this path must contain edges of the polygonal obstacles and of segments connecting the starting and ending points to polygon vertices.

The pathfinding illustrated here is for a game character represented as a single point located at the camera eye point. A complication is that if the eye point is allowed to be close to the sides of obstacles (or walls), the geometry can intersect the near plane of the view frustum, thereby allowing the observer to see inside the obstacle. To avoid this problem, the game character may be thought of as a circular disk within the blueprint; for example, the disk can be the projection of a bounding cylinder or capsule for the character. Unobstructed navigation through the room requires a path for the disk center to traverse without the disk intersecting the polygonal obstacles. An equivalent formulation shrinks the disk to a point and grows the obstacles by the radius of the sphere. The point must follow a path through the modified environment

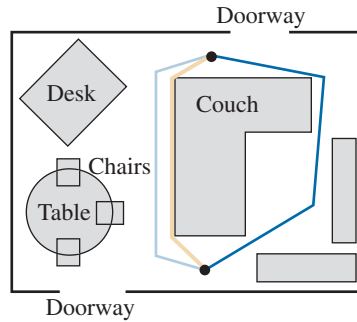


Figure 8.30 Movement paths between two points in a room. The black dots indicate the starting and ending points. The light blue and dark blue paths are randomly selected. The taupe path is the shortest path.

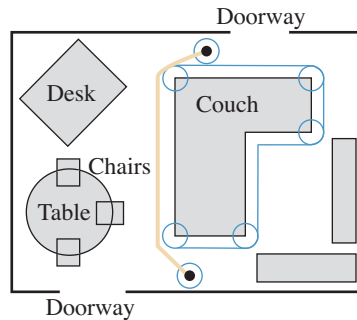


Figure 8.31 The starting and ending points both have disks indicating the size of the game character. The couch is enlarged by the radius of the disk. The shortest path hugs the boundary of the enlarged couch.

without passing through the enlarged obstacles, as illustrated in Figure 8.31. The appropriate choice for radius depends on the camera view frustum parameters.

The enlarged polygonal objects have boundaries that consist of line segments and circular arcs. The pathfinding algorithm becomes much more complicated for such objects compared to polygonal ones. It is not necessary to be precise and use a disk to represent the game character. Instead, we may use a regular polygon of appropriate size. The enlarged polygonal objects are then also polygonal objects,

so the pathfinding algorithm remains fairly simple. But even this can be tedious to implement. If the near-plane problem does show up for some objects, an alternative is to have the artist generate nondisplayable geometry to be used for creating the obstacle, but that geometry amounts to a bounding volume for the actual object and the bounding volume is larger than need be. The path will hug the polygon of projection of the bounding volume, but not touch the actual obstacle projection.

Obstacles in a room are optional objects. Whether an object is an obstacle is the choice of the artist. Rooms may have other objects that are not tagged as obstacles, in which case the pathfinding system will ignore them when building blueprints. For example, if a room has a chandelier that is tall enough not to obstruct the game character's path, there is no reason to make it an obstacle.

Optionally, each room also has obstacles called *transporters*. These are objects that house the connections between levels, to be discussed later. For example, the transporter might be the geometry for an elevator shaft or a stairwell.

8.5.2 MOVING BETWEEN ROOMS

Constructing the visibility graph for the entire indoor-outdoor environment can be very expensive. Partitioning of the levels into rooms, each room having its own visibility graph, makes the problem less computationally expensive. To assist with the pathfinding between rooms, each doorway has an associated *waypoint*. If the starting point is in one room and the ending point is in an adjacent room, the pathfinder will construct two subpaths, one from the starting point to the waypoint of the doorway connecting the two rooms and one from the waypoint to the ending point. Figure 8.32 illustrates this. The construction of the subpath in Room A is based only on the visibility graph for Room A. Similarly, the subpath in Room B is based only on the visibility graph for Room B. The introduction of waypoints may be viewed as a graph decomposition, each subgraph small enough to allow fast pathfinding within it.

If a doorway between two rooms is very wide, a path connecting the rooms will go through the waypoint, even if the starting and ending points are close to each other through the doorway. Figure 8.33 illustrates this. The wide doorway is represented as two smaller doorways, one drawn in light blue and one drawn in taupe. Each of the smaller doorways has a waypoint. The gray path from start to end is the one that would have been taken with a single doorway and waypoint. The dark blue path from start to end is the one taken with the two waypoints. The introduction of invisible doorways and more waypoints is also a useful technique for partitioning the abstract rooms of the outside level.

8.5.3 MOVING BETWEEN LEVELS

The previous discussion was in the context of pathfinding within a room and between rooms on the same level. The buildings may consist of multiple levels, and the game

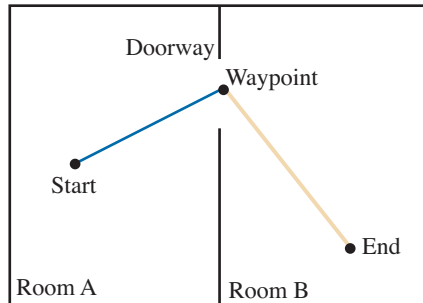


Figure 8.32 A path between adjacent rooms consists of two subpaths meeting at the waypoint of the doorway connecting the two rooms.

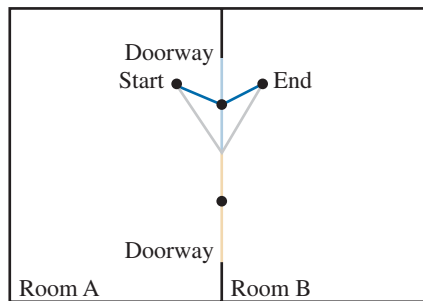


Figure 8.33 A path connecting two points, which is required to go through a single waypoint for a wide doorway, is drawn in gray. A shorter path connecting the two points is shown based on the doorway being represented as multiple doorways, each with its own waypoint.

character will be allowed to move from one level to another within a building. The geometric objects over which the game character may travel are called *level connectors*, for example, stairs, ramps, and elevators. To keep the pathfinding from becoming overly complex, you might want to impose the constraint that no obstacles occur on level connectors, but it is possible to design the pathfinding system to handle obstacles within connectors. Assuming you have chosen not to have obstacles inside the level connectors, the paths for the level connectors may be constructed as a preprocessing step. In gaming terminology, the game character is said to move on *rails* when traveling through the connector. Observe the similarity between a waypoint

between two rooms and a rail between two levels. Both support the decomposition of the total visibility graph for the environment into disjoint subgraphs.

If the game character is on one level and the user selects a connector to traverse to another level, the pathfinding can move the game character first to a *transporter* housing the connector and then along the rail to the other level. For example, a transporter would represent an elevator shaft. The transporter can have one or more *hatches* that represent the entrance into the transporter. An automated pathfinding system can generate a path from the game character (the source) to one of the hatches (the destination). A subpath from the hatch to the start of the rail can be generated and appended to the current path, and then the rail itself is appended to the path. The game character may then be moved along this compound path to arrive at its desired destination. One problem, though, is that once the game character arrives at a hatch point via an automatically generated path, he or she is ready to enter the transporter and follow the rail to a connecting level. The problem, though, is that there may be multiple levels. For example, if a game character arrives at a hatch that is an entrance to a stairwell in a building, he or she has multiple levels in the building to travel to. Thus, a transporter must represent multiple connections between levels. The pathfinding system needs to have support for queries about which levels are available to proceed to. The application can present the possibilities to the user, perhaps using a GUI component such as a pop-up menu. The user selects one of the possibilities and the pathfinding system appends the corresponding subpaths to the current path of the game character.

8.5.4 MOVING THROUGH THE OUTDOOR ENVIRONMENT

The outdoor environment may be thought of as a single level with a single room. As such, it may contain obstacles such as bushes, trees, benches, and grass patches with keep-off-the-grass signs. The buildings themselves may be considered obstacles, but with doorways (and waypoints) connecting the indoors and outdoors. If the outdoor environment is large enough, you may want to consider using waypoints to decompose it into smaller pieces to keep the visibility graph updates inexpensive. The node tagging system and a postprocessing tool can be used to support this. If the outdoor level is to be connected to a building level, the entranceway connecting inside and outside would be represented as a level connector, and the entranceway itself is a transporter object. The rail is a line segment passing through the entranceway.

8.5.5 BLUEPRINTS

Automatic pathfinding through a general 3D environment is an expensive proposition, typically relying on collision detection to prevent the game character from walking through obstacles or walls. Generating such paths without regard to mesh geometry, topology, or the organization of objects in the environment places a lot of

burden on the programmer to get a general system to work. I consider this an intractable approach. It is better to make some simplifying assumptions and to place some of the burden on the artists to structure the scenes and tag them with information to support an automatic pathfinding system.

As described in this section, the pathfinding is reduced to a 2D problem. Imagine a level of interconnected rooms, all with vertical walls. The projection of the walls leads to 2D polygons for the room boundaries. The obstacles in the room also project to 2D polygons. The result of the projection is what I call a *blueprint*, similar to the blueprints that you see for architecting a building. See Figure 8.29 for a typical blueprint for a single room with some obstacles.

Pathfinding in 2D is a lot simpler to implement than in 3D. In 2D, the environment consists of a collection of polygons, a source point (the game character's current location), and a destination point (the game character's desired location). This topic has been studied extensively in the field of computational geometry and amounts to constructing a *visibility graph* of the polygon vertices, source point, and destination point.

Keep in mind that the actual scene is still 3D, so the blueprints for a level must be generated somehow. A scene graph must be created by the artist and its nodes tagged subject to various constraints. Room walls, obstacles, transporters, and other objects of interest must be created and tagged in order to facilitate blueprint construction.

8.5.6 VISIBILITY GRAPHS

The heart of the pathfinding system is the concept of *visibility graphs*. Given a source point, a destination point, and a set of polygon obstacles, the goal is to compute the shortest path from the source to the destination, all the while avoiding passing through the obstacles. See Figure 8.30 for the typical scenario. The shortest path must be a polyline whose endpoints are the source and destination and whose interior points are polygon vertices.

Static Visibility

The vertices of the polygon obstacles are the graph vertices for what is called the *static visibility graph*. The modifier *static* refers to the fact that the polygons never move during the pathfinding. Thus, the portion of the total visibility graph corresponding to the polygon vertices can be precomputed and used at run time.

Figure 8.34 shows two triangle obstacles. The graph vertices are $\{\mathbf{V}_i\}_{i=0}^5$. The graph edges are abstractly represented by a 6×6 adjacency matrix. The matrix is symmetric because the edges are undirected. The matrix entry m_{ij} is set to 1 if $\mathbf{V}_i$ and $\mathbf{V}_j$ can see each other; that is, their view of each other is unobstructed by a triangle edge. Otherwise, the entry is set to 0. By convention, a vertex cannot see itself, so the diagonal of the adjacency matrix has all zeros. Also by convention, two consecutive

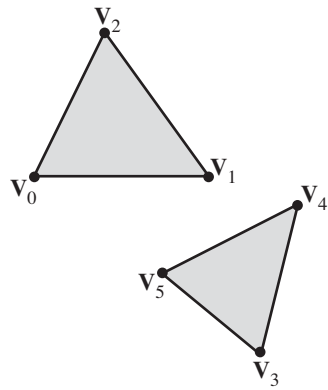


Figure 8.34 Two triangle obstacles. The static visibility graph has six vertices and 12 undirected edges out of 15 possible undirected edges.

Table 8.3 The adjacency matrix for the visibility graph of Figure 8.34.

	V ₀	V ₁	V ₂	V ₃	V ₄	V ₅
V ₀	0	1	1	1	1	1
V ₁	1	0	1	0	1	1
V ₂	1	1	0	0	1	0
V ₃	1	0	0	0	1	1
V ₄	1	1	1	1	0	1
V ₅	1	1	0	1	1	0

polygon vertices can see each other. The adjacency matrix for the visibility graph of the triangles in Figure 8.34 is shown in Table 8.3.

The pseudocode for computing the static visibility graph is

```

VisibilityGraph G;
G.Vertices = <set of polygon vertices>;
for each P in G.Vertices do
{
    for each Q in G.Vertices do
    {
        bool isGraphEdge = true;
    }
}

```

```

for each polygon edge E do
{
    if E occludes P from seeing Q then
    {
        isGraphEdge = false;
        break;
    }
}
if (isGraphEdge)
{
    G.Edges.Insert(P,Q);
}
}

```

For example, in Figure 8.34, the triangle edge $\langle V_4, V_5 \rangle$ occludes the view of V_2 from V_3 , so the visibility graph edges do not include $\langle V_2, V_3 \rangle$. The adjacency matrix entry m_{23} is therefore 0.

Dynamic Visibility

The source point is the location of the game character. The destination point is selected by some means. The shortest path connecting the source and destination is a polyline whose endpoints are the source and destination and whose interior points are polygon vertices. The source and destination points will change frequently, thus affecting what is visible to them. The static visibility graph must be updated to include the source and destination. The resulting graph is called the *dynamic visibility graph*. In the actual application, the dynamic visibility graph is computed temporarily by inserting visibility edges related to the source and destination, but once the shortest path query is executed, those edges are removed and the static visibility graph is restored.

Figure 8.35 shows the triangle obstacles of Figure 8.34, a source point, and a destination point. The shortest path between them is also shown. The adjacency matrix representation of the graph edges is expanded to accommodate the source and destination, so it is now an 8×8 matrix. The source point is denoted **S** and the destination point is denoted **D**. Table 8.4 shows the expanded adjacency matrix representation.

The shortest path is the sequence of points $\{S, V_5, V_1, D\}$. In this example, the source cannot see the destination. In a situation where it can see the destination, then the shortest path is just the line segment connecting the source and destination.

In theory, the adjacency matrix representation for the visibility graph is intuitive, but in practice the storage requirements are excessive. An implementation should use dynamically resizable containers. I tend to use an `std::vector` data structure to store

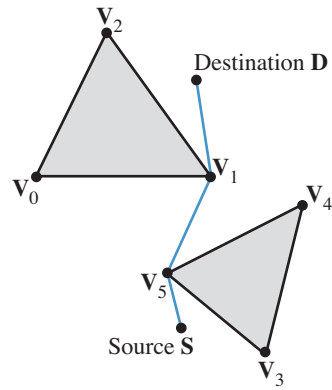


Figure 8.35 Triangle obstacles, source and destination points, and the shortest path connecting those points.

Table 8.4 The expanded adjacency matrix for the graph of Figure 8.35.

	V_0	V_1	V_2	V_3	V_4	V_5	S	D
V_0	0	1	1	1	1	1	1	0
V_1	1	0	1	0	1	1	0	1
V_2	1	1	0	0	1	0	0	1
V_3	1	0	0	0	1	1	1	0
V_4	1	1	1	1	0	1	0	1
V_5	1	1	0	1	1	0	1	0
S	1	0	0	1	0	1	0	0
D	0	1	1	0	1	0	0	0

the graph vertices. My choice of container for the graph edges depends on how large the graphs will be in my applications.

Dijkstra's Algorithm

At the core of the graph data structure is the support for computing the shortest paths from a single source vertex to all the other graph vertices. The standard algorithm for

doing this is *Dijkstra's algorithm*, which is essentially a relaxation scheme to compute the minimum weight paths in a directed graph. The details are provided in the book [CLR90].

Each vertex has associated with it two values: an *estimate* of the distance from the source to that vertex, and a *predecessor* in the current path from the source to the vertex that estimates the shortest path. All vertices are stored in an array, so the predecessor is stored as an index into that array. Initially, all estimates are set to infinity, except for the source vertex whose estimate is set to zero, and the predecessors are set to -1 (invalid index). Each edge has associated with it the length of the segment connecting the two vertices that make up the edge.

The vertex with the minimum estimate is removed from the array; call it $V[\text{min}]$. There is a candidate shortest path from source S to $V[\text{min}]$ with path length given by $V[\text{min}].\text{estimate}$. The path is

$$S, \dots, V[V[\text{min}].\text{predecessor}], V[\text{min}]$$

The vertices adjacent to $V[\text{min}]$ are processed. Each adjacent vertex $V[\text{adj}]$ also has a candidate shortest path,

$$S, \dots, V[V[\text{adj}].\text{predecessor}], V[\text{adj}]$$

with a path length $V[\text{adj}].\text{estimate}$. If the predecessor for $V[\text{adj}]$ is not $V[\text{min}]$, it is possible that the following path is a shorter route from S to $V[\text{adj}]$:

$$S, \dots, V[\text{min}], V[\text{adj}]$$

If it is, the predecessor for $V[\text{adj}]$ is updated to $V[\text{min}]$. The update will occur only when

$$V[\text{min}].\text{estimate} + E[\text{min}][\text{adj}].\text{length} < V[\text{adj}].\text{estimate}$$

and, in this case, the adjacent vertex's estimate is set to the new smaller estimate (the left-hand side of the inequality).

The selection of the minimum-estimate vertex is the main bottleneck in the algorithm. If the number of graph vertices is relatively small, a linear search for the vertex with the minimum estimate is a reasonable choice. If the number of graph vertices is large, the linear search should be replaced with a priority queue, which greatly reduces the cost for the search.

The previous discussion was about computing the shortest paths from a single source to all other vertices. The pathfinding system needs access to all the shortest paths between vertices. Dijkstra's algorithm is executed for each graph vertex and the results are stored in an $n \times n$ array, where n is the number of vertices. Each entry of the array stores the distance from one vertex to the other and stores the predecessor for that path. If later your data sets lead to very large values of n , and if physical memory is not large enough to store these arrays, either they must be stored on

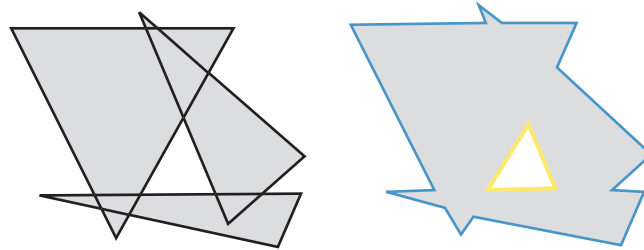


Figure 8.36 An obstacle projection consisting of three triangles. The envelope is marked in blue and is selected to be the 2D obstacle polygon for blueprints. The inner polygon marked in yellow is not necessary for the pathfinding, but is considered to be part of the silhouette.

disk (bottleneck in disk I/O operations) or they must be computed only as needed (bottleneck in CPU computation). Preferably in this situation, the obstacle polygons should be reduced in size by not requiring an exact fit of the projected obstacle data. For example, you could use oriented bounding boxes or discrete oriented polytopes as proxies for the obstacle envelope. This will keep n to a reasonable size (per room).

8.5.7 ENVELOPE CONSTRUCTION

The 2D obstacle polygons in the blueprints are constructed by projecting the 3D obstacles in the scene onto the ground plane and locating the outer edges of the projection. I call this set of edges the *envelope* of the projection; it is a subset of the boundary of the *silhouette* of the projection. The silhouette includes additional polygons that are nested inside the envelope. Figure 8.36 illustrates the projection of an object consisting of three triangles. The outer blue polygon in the figure is what must be constructed from the projected triangles. This will involve computing intersections of pairs of triangle edges and selecting subedges that make up the outer polygon. This is nearly a union operation on polygons. A full union operation will compute the silhouette.

Projection Graph

The edges of all the triangles of the 3D obstacle are projected onto the ground plane. The 2D vertices and edges form an abstract undirected graph, the vertices being the graph nodes and the edges being the graph arcs. The outer polygon, though, involves subedges whose endpoints are either the original vertices or points of intersection

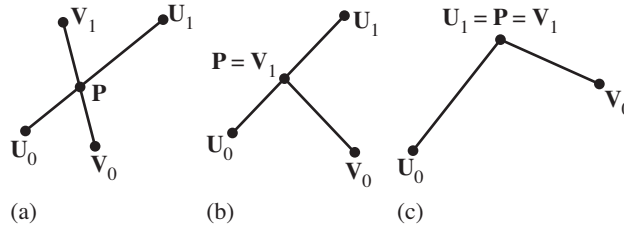


Figure 8.37 Three distinct configurations for the intersection of two edges.

between edges. Thus, it is necessary to compute all edge-edge intersections and introduce new vertices and new edges into the graph, the final result called the *projection graph*.

Given graph vertices U_0, U_1, V_0 , and V_1 and graph edges $\langle U_0, U_1 \rangle$ and $\langle V_0, V_1 \rangle$, if the two edges intersect at P , then the graph must be updated to incorporate this information. The update depends on how the edges intersect. Figure 8.37 illustrates the three distinct configurations.

In Figure 8.37 (a), the point P is inserted as a new vertex in the graph. The edges $\langle U_0, U_1 \rangle$ and $\langle V_0, V_1 \rangle$ are removed from the graph. Four new edges are inserted in the graph, namely, $\langle U_0, P \rangle$, $\langle U_1, P \rangle$, $\langle V_0, P \rangle$, and $\langle V_1, P \rangle$. In Figure 8.37 (b), the intersection point happens to already be a graph vertex, so no new vertices must be inserted in the graph. The edge $\langle U_0, U_1 \rangle$ is removed and two new edges are inserted, namely, $\langle U_0, V_1 \rangle$ and $\langle U_1, V_1 \rangle$. Finally, Figure 8.37 (c) shows two graph edges intersecting at a graph vertex, so the graph does not need to be updated. This case is expected frequently in the construction because two edges sharing a vertex in the 3D obstacle will project to two edges sharing a vertex in the ground plane. However, it is possible that the configuration of Figure 8.37 (c) occurs from two nonadjacent edges in the 3D obstacle. This happens when two distinct vertices in the 3D obstacle project to the same point on the ground plane.

Rational Arithmetic for Edge Intersections

Theoretically, the projection graph construction is a straightforward use of graph data structures, with the various insertions and removals as indicated in the previous paragraph. In practice, though, a few problems must be handled. As is well known in computational geometry, round-off errors in floating-point arithmetic can wreak havoc with theoretical algorithms that are guaranteed to work when the arithmetic is exact. It is essential for the pathfinding system that the envelope be a correctly formed simple polygon. Round-off errors can lead to failure to find an intersection when there is one and to finding an intersection when there is not one. The side

effects usually manifest themselves in generating incorrect topology for the vertex-edge mesh represented by the graph, which makes it difficult to construct special sequences of edges in the graph. In fact, to avoid this, the envelope construction is performed using exact rational arithmetic. The 2D projected vertices have floating-point components, each component exactly representable as a rational number. The intersections of edge pairs are computed using rational arithmetic.

The rational arithmetic implementation is in the Foundation library directory named `RationalArithmetic`. The implementation details are not presented here but are based on Donald Knuth's discussion of exact arithmetic in [Knu73, Chapter 4]. The implementation assumes a fixed-size integer (for numerator and for denominator), which in itself must be understood when using the classes. Generally, as the number of arithmetic operations increases when manipulating rational numbers, the storage requirements increase. An arbitrary-precision arithmetic library will provide dynamic reallocation, but a fixed-precision library will not and can only report when there is overflow.

The all-pairs, edge-edge intersection tests have been structured to avoid overflow. To motivate what is done, consider the intersection of two edges $\langle \mathbf{U}_0, \mathbf{U}_1 \rangle$ and $\langle \mathbf{V}_0, \mathbf{V}_1 \rangle$. The line segments are parameterized by $\mathbf{U}_0 + s(\mathbf{U}_1 - \mathbf{U}_0)$ for $0 \leq s \leq 1$ and $\mathbf{V}_0 + t(\mathbf{V}_1 - \mathbf{V}_0)$ for $0 \leq t \leq 1$. A point of intersection is computed by equating these:

$$\mathbf{U}_0 + s(\mathbf{U}_1 - \mathbf{U}_0) = \mathbf{V}_0 + t(\mathbf{V}_1 - \mathbf{V}_0)$$

solving for s and t , and verifying the parameters are in the interval $[0, 1]$. The solution is

$$s = \frac{(\mathbf{V}_0 - \mathbf{U}_0) \cdot (\mathbf{V}_1 - \mathbf{V}_0)^\perp}{(\mathbf{U}_1 - \mathbf{U}_0) \cdot (\mathbf{V}_1 - \mathbf{V}_0)^\perp}, \quad t = \frac{(\mathbf{V}_0 - \mathbf{U}_0) \cdot (\mathbf{U}_1 - \mathbf{U}_0)^\perp}{(\mathbf{U}_1 - \mathbf{U}_0) \cdot (\mathbf{V}_1 - \mathbf{V}_0)^\perp}$$

where $(x, y)^\perp = (y, -x)$. The numerator involves differences of vectors. The subtraction of two components is a rational expression of the form $a/b - c/d = (ad - bc)/(bd)$, where a, b, c , and d are n -byte integers. The product of two n -byte integers requires $2n$ bytes of storage. The sum or difference of two n -byte integers requires, in worst case, $n + 1$ bytes—the extra byte to handle a 1-bit overflow. In total, the result requires $2n + 1$ bytes in worst case. The dot product operation is of the form $ab + cd$, where a, b, c , and d are rational numbers, each using integers of size $2n + 1$. The result requires $8n + 5$ bytes in worst case. The products $a * b$ and $c * d$ double the $2n + 1$ requirement. The sum of these two rational numbers induces more multiplications, doubling the requirement again, and one additional byte is needed to handle overflow in the sum in the numerator. The numbers s and t are ratios of rational numbers, each requiring at most $8n + 5$ bytes, which includes yet another multiplication. The maximum storage requirement is now $16n + 10$. The point of intersection itself has rational expressions of the form $(a/b) + (e/f) * (c/d)$, where a, b, c , and d are n -byte integers and $s = e/f$ is a rational number where e and f are $16n + 10$ -byte integers. The result requires $18n + 11$ bytes in worst case.

The `TInteger` and `TRational` classes have a template parameter, `N`. A `TInteger` object has a fixed size of $4N$ bytes. The maximum floating-point number is essentially 2^{128} , which requires a 16-byte numerator, so $n = 16$ and $18n + 11 = 299$. However, the likelihood that the artist generated models having extremely large floating-point values is small, so in practice the $18n + 11$ is too large. For typical artist-generated models, $n = 64$ is a reasonable choice. The larger n is, the longer the arithmetic operations take. To reduce the computation time, keep n relatively small. In the event overflow occurs for a data set, this will be trapped when running in Debug mode. At that time you can increase the size of n .

The problem with rational number size is potentially compounded if an edge is intersected by multiple edges. In the previous example, if P is the intersection of edges $\langle U_0, U_1 \rangle$ and $\langle V_0, V_1 \rangle$, and the point lies in the interior of the first edge, you could insert the subedges $\langle U_0, P \rangle$ and $\langle P, U_1 \rangle$ into the graph and test them for intersections with other edges. Using rational arithmetic, the storage requirements increase to compute P . If you were then to use P in computing an intersection of another edge with $\langle U_0, P \rangle$, this intersection point requires yet even more memory than P . To avoid this situation, all edge-edge intersection tests are executed using the initial edges in the graph. This guarantees a bounded amount of memory for the intersection points.

At the highest level of abstraction, each edge is tested for intersection against the other edges, and a sorted list of parameter values for the intersection points is maintained. Once all the edges are processed, the original projection graph is destroyed, a new one created, and all the vertices and subedges implied by the sorted lists of parameter values are inserted into the new graph. It is this new graph that is processed to find the outer edges forming the obstacle envelope.

Sort-and-Sweep for Fast Intersections

A double loop for testing all pairs of edges is straightforward to implement, but very inefficient and costly in execution time. This approach has no localization to it; that is, edges are tested for intersection regardless of how close or how far apart they are. The approach has asymptotic order $O(n^2)$ for n edges.

A better approach is to use spatial coherence to help reduce the number of pairs of edges to test for intersection. A simple and effective one is based on a *sort-and-sweep algorithm*. The idea is to project the axis-aligned bounding boxes (AABBs) onto the x -axis, obtaining a collection of intervals. The interval endpoints are sorted, all the time maintaining the information about which points start the intervals and which points end the intervals. The sweep phase is to traverse the sorted intervals. For each pair of overlapping intervals, the AABBs are tested for overlap in the y -direction. If there is overlap, the two corresponding edges are tested for intersection. If there is no overlap, the edges cannot intersect and there is no reason to test them for intersection. The detailed description is given in Section 8.3.3, which is based on material from [Bar01], but the algorithm originated in general dimensions in [PS85].

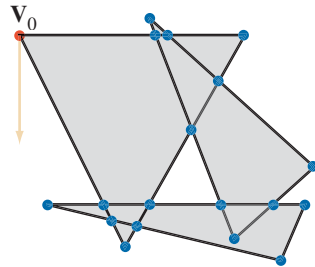


Figure 8.38 The projection graph for the three triangles of Figure 8.36. The 19 vertices are drawn in blue, except for the starting vertex of the outer boundary traversal, which is drawn in red. The vertices are interconnected by 29 edges drawn in black. The *current direction* is drawn in taupe.

Locating the Outer Edges

Once all edge-edge intersections are located and the new projection graph built, we are ready to traverse the graph to locate the outer edges that make up the envelope. To illustrate, the projection graph of the three triangles in Figure 8.36 is shown in Figure 8.38. The idea is to start at a vertex on the outer boundary and follow edges while remaining on the outer boundary. A simple choice for the starting vertex is the one with minimum x -value. If there are multiple points attaining the minimum, then choose the vertex with the minimum y -value of all these. The red vertex in Figure 8.38 is the starting vertex for the traversal.

The taupe arrow in Figure 8.38 indicates the current direction of traversal from the previous vertex on the outer edge. Initially, there is no previous vertex, but the direction is selected as if there were one with minimum x -value but larger y -value.

Figure 8.39 shows the first step of the algorithm. The current vertex is labeled V_0 . It has two edges sharing it, the adjacent vertices labeled A_0 and A_1 . The outer edge starting at V_0 is chosen to be the edge that has no other edges between it and the current traversal direction. The edge $\langle V_0, A_0 \rangle$ is eliminated as a candidate because the edge $\langle V_0, A_1 \rangle$ is between it and the current direction (in taupe). The edge $\langle V_0, A_1 \rangle$ is the chosen outer edge because no other edge sharing V_0 is between this edge and the current direction vector. Figure 8.39 shows the selected outer edge in red.

The concept of one vector “between” two other vectors needs to be quantified mathematically for computing. Figure 8.40 shows the two configurations to handle. Think of the test for betweenness in terms of the cross product of the vectors as if they were in 3D with z -components of zero, and apply the right-hand rule. Define the 2D vectors $D_0 = A_0 - V$, $D_1 = A_1 - V$, and $D = P - V$. Define the 3D vectors $E_0 = (D_0, 0)$, $E_1 = (D_1, 0)$, and $E = (D, 0)$; that is, the vectors have zero for their z -components.

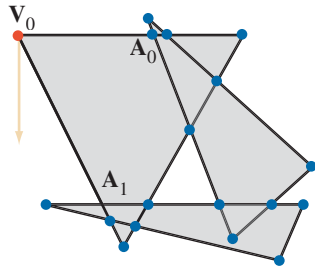


Figure 8.39 The current vertex is V_0 and has two adjacent vertices A_0 and A_1 .

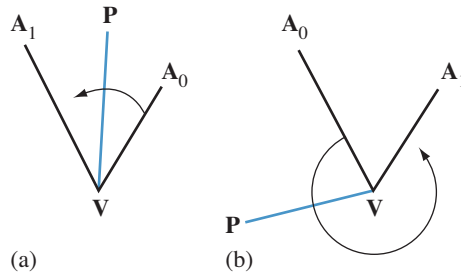


Figure 8.40 (a) The vertex V is a convex vertex relative to the other vertices A_0 and A_1 . (b) The vertex V is a reflex vertex relative to the other vertices. In both cases, a vector $P - V$ between vectors $A_0 - V$ and $A_1 - V$ is shown in blue.

In the case where V is convex with respect to its neighbors, D is between D_0 and D_1 when the cross products $E \times E_1$ and $E_0 \times E$ both have positive z -components. That is, if you put your right hand in the direction E with your thumb up (out of the plane of the page), and rotate your fingers toward your palm (rotating about your thumb), you should reach E_1 . Similarly, if you put your right hand in the direction E_0 and rotate your fingers toward your palm, you should reach E . Note that

$$E \times E_1 = (0, 0, D \cdot D_1^\perp), \quad E_0 \times E = (0, 0, D_0 \cdot D^\perp)$$

The test for strict betweenness is therefore

$$D_0 \cdot D^\perp > 0 \quad \text{and} \quad D \cdot D_1^\perp > 0 \quad (8.3)$$

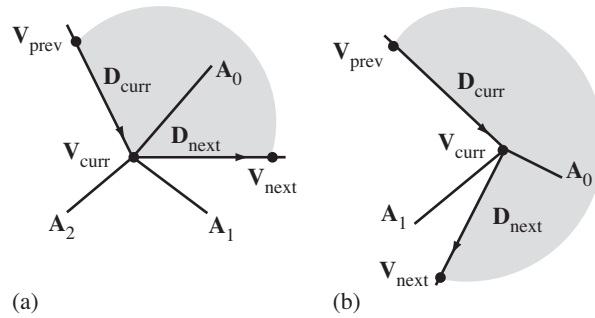


Figure 8.41 (a) The current vertex is convex with respect to the previous vertex and the next vertex. (b) The current vertex is reflex with respect to the previous vertex and the next vertex. The gray regions are part of the projection obstacle whose outer boundary is sought.

In the case where V is reflex with respect to its neighbors, D is between D_0 and D_1 (in that order) when it is *not between* D_1 and D_0 (in that order). This is the negation of the test in Equation (8.3) with the roles of D_0 and D_1 swapped, and with the strict containment condition, namely,

$$D_1 \cdot D^\perp < 0 \quad \text{or} \quad D \cdot D_0^\perp < 0 \quad (8.4)$$

The betweenness measurements show up when visiting all the adjacent vertices of the current vertex V_{curr} . The previous vertex found on the outer edge is denoted V_{prev} . When the current vertex is the initial vertex V_0 of minimum x -value, the current direction $D_{\text{curr}} = (0, -1)$ and implies a previous vertex of $V_{\text{prev}} = V_0 - D_{\text{curr}}$. The next vertex, denoted V_{next} , is the adjacent vertex that is the candidate for the next vertex on the outer edge. Figure 8.41 illustrates the configuration for a convex and a reflex current vertex.

When a current vertex is selected, its adjacent vertices are searched to determine which will be the next vertex on the outer edge. One of the adjacent vertices will be the previous vertex, which is rejected as the candidate for the next vertex. Clearly, this is necessary to avoid backtracking on an edge. The projection of vertical faces of an obstacle can also produce adjacent vertices that are in the direction opposite that of the current directions. These also are rejected as candidates for the next vertex, again to avoid backtracking. The first found adjacent vertex that is not the previous vertex and is not in the direction opposite the current direction is chosen as V_{next} . Figure 8.41 illustrates the situation when the next vertex has been chosen.

The current vertex has other adjacent vertices to be tested if they should become the next vertex. In the convex case, Figure 8.41 (a) shows three adjacent vertices,

A_0 , A_1 , and A_2 , of V_{curr} to be processed. The adjacent vertex A_0 is rejected because the edge to it is inside the current region around which we are building the outer boundary. Both A_1 and A_2 are candidates for the next vertex because the edges to them are outside the current region. If A_1 is visited first, V_{next} is set to that vertex. When A_2 is visited, it will become yet the next vertex. If A_2 is visited first, it will become the next vertex. Then A_1 is visited, but it will be inside the current region and is therefore rejected. In either case, the vector $D = A_i - V_{\text{curr}}$ for $i = 1$ or $i = 2$ is between the vectors $V_{\text{prev}} - V_{\text{curr}} = -D_{\text{curr}}$ and $V_{\text{next}} - V_{\text{curr}} = D_{\text{next}}$, in that order. The algebraic test for this is an application of Equation (8.4)—the current vertex is reflex relative to the *outside* region:

$$D_{\text{next}} \cdot D^\perp < 0 \quad \text{or} \quad D \cdot (-D_{\text{curr}})^\perp < 0$$

or equivalently,

$$D_{\text{curr}} \cdot D^\perp < 0 \quad \text{or} \quad D_{\text{next}} \cdot D^\perp < 0$$

In the reflex case, Figure 8.41 (b) shows two adjacent vertices, A_0 and A_1 , of V_{curr} to be processed. The adjacent vertex A_0 is rejected because the edge to it is inside the current region around which we are building the outer boundary. The adjacent vertex A_1 becomes the next vertex because the edge to it is outside the current region. The vector $D = A_i - V_{\text{curr}}$ is between $-D_{\text{curr}}$ and D_{next} . The algebraic test for this is an application of Equation (8.3)—the current vertex is convex relative to the *outside* region:

$$-D_{\text{curr}} \cdot D^\perp > 0 \quad \text{and} \quad D \cdot D_{\text{next}}^\perp > 0$$

or equivalently,

$$D_{\text{curr}} \cdot D^\perp < 0 \quad \text{and} \quad D_{\text{next}} \cdot D^\perp < 0$$

The processing of the outer edges and the vertices on them continues until the starting vertex is reached.

Performance Testing

I tested the algorithm on a large data set to see how the exact rational arithmetic performs. Figure 8.42 shows a rendered 3D face with 1330 vertices and 2576 triangles. Figure 8.43 shows the projections and envelopes for two different orientations of the face. The program was run on two different machines. The results are listed in Table 8.5. The times for the orientation in Figure 8.43 (a) are significantly shorter than those for Figure 8.43 (b) because image (a) is essentially the graph of a function (a height field, as it were). No pair of edges intersects at interior points, so the only exact

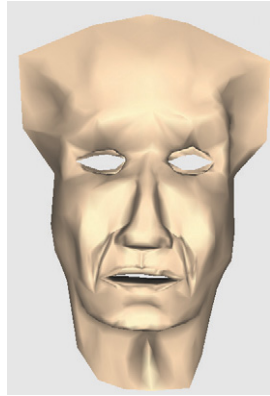


Figure 8.42 A 3D face with 1330 vertices and 2576 triangles.

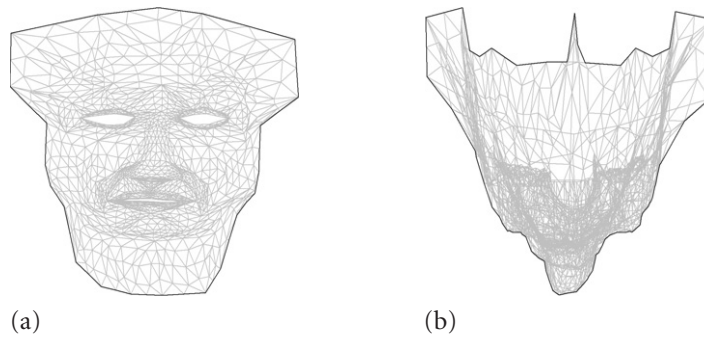


Figure 8.43 The projections and envelopes of the face of Figure 8.42. The face appears stretched because the 2D display application did not adjust for the aspect ratio of the original data when displayed in 3D.

arithmetic costs are in the location of the outer edge. The orientation of image (b) is such that quite a few pairs of edges intersect at interior points.

The envelope extraction was implemented without any assumptions on the obstacle other than that the projection is a connected region on the ground plane. Obstacles with multiple connected components can be split by the artist into multiple obstacles, each producing a single connected component on the ground plane.

Table 8.5 Run times for the envelope construction, measured in seconds.

<i>Orientation</i>	<i>Configuration</i>	<i>Intel Pentium 3 (1GHz)</i>	<i>AMD Athlon XP 2800+ (2.08GHz)</i>
Figure 8.43 (a)	Debug	40.178	19.594
Figure 8.43 (a)	Release	13.219	5.609
Figure 8.43 (b)	Debug	252.383	117.344
Figure 8.43 (b)	Release	63.541	26.907

If you are willing to require the artists to create the obstacles as *manifold meshes*, the run time for the extraction can be significantly reduced. A manifold mesh is an orientable mesh for which each edge is shared by at most two triangles. In this case, the only edges that need to be projected and processed are those edges shared by a single triangle (the mesh “boundary” edges) and those edges shared by two triangles, one triangle with a nonnegative normal vector component in the direction of the world up vector and one triangle with a nonpositive normal vector in the direction of the world up vector. If $\mathbf{N}_0$ and $\mathbf{N}_1$ are the outer-pointing triangle normals, and if $\mathbf{U}$ is the world up vector, the common edge is projected and processed when $\mathbf{U} \cdot \mathbf{N}_0 \geq 0$ and $\mathbf{U} \cdot \mathbf{N}_1 \leq 0$. These edges are referred to as *contour edges*, a subset of which are the silhouette edges (and the envelope edges are a subset of the silhouette edges).

8.5.8 BASIC DATA STRUCTURES

The basic data structures that make up an environment are implemented by the classes Environment, Level, Room, Boundary, HeightField, Doorway, Obstacle, Transporter, and Connector. The objects themselves are dynamically allocated and interconnected via pointer references, forming a union of graphs and multigraphs. Any graph of dynamic objects that contains cycles requires some notion of ownership for the purpose of deleting the objects. Figure 8.44 illustrates the class interrelationships.

The black arrows indicate that an object of the class at the beginning of the arrow references an object of the class at the end of the arrow. The object for the class at the beginning is responsible for deleting the objects it references. The blue arrows indicate that an object of the class at the beginning of the arrow references an object of the class at the end of the arrow. However, the object of the class at the beginning is only sharing the other object and has no responsibility, so delete it. The named rectangles in Figure 8.44 and the black arrows imply a minimum spanning tree for the graph of actual objects, which is a directed acyclic graph.

The implementation details are tedious, but tractable. I leave these details up to you.

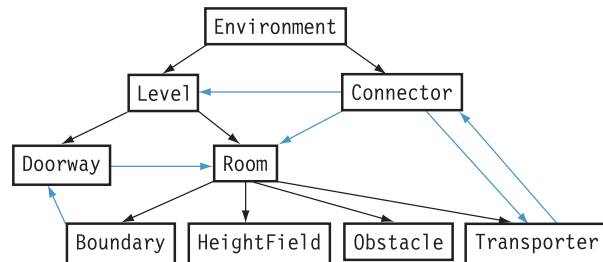


Figure 8.44 The interrelationship of classes for the basic data structures in the pathfinding system.

8.5.9 EFFICIENT CALCULATION OF THE VISIBILITY GRAPH

For a single room in the environment, the static visibility graph vertices consist of the waypoints of the doorways, the vertices of the obstacle polygons, and the vertices of the transporter polygons. An edge in the visibility graph is a pair of vertices for which the line segment connecting them does not intersect any polygon edge at an interior point of the polygon edge. That is, the line of sight from one vertex to the other is unobstructed by a polygon edge. The pseudocode for the naive approach to computing graph edges is

```

VisibilityGraph G;
G.Vertices = {waypoints, obstacle vertices, transporter vertices};
for each P in G.Vertices do
{
    for each Q in G.Vertices do
    {
        bool isGraphEdge = true;
        for each obstacle or transporter polygon edge E do
        {
            if segment (P,Q) intersects E at an interior point then
            {
                isGraphEdge = false;
                break;
            }
        }
        if (isGraphEdge)
        {
            G.Edges.Insert(P,Q);
        }
    }
}

```

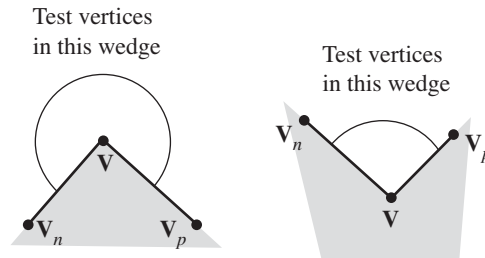


Figure 8.45 Graph vertices visible to V must lie in the wedge defined by V and its two neighbors.

For n graph vertices, this is an order $O(n^3)$ algorithm. The asymptotic order can be reduced, but the complexity of the algorithms increases greatly. The possibilities include the use of arrangements, which is an $O(n^2)$ algorithm, or the use of an output-sensitive method, which is $O(n \log n + m)$, where m is the number of graph edges. Both are referenced in the book [O'R98, Section 8.2.2] with only a single paragraph of discussion.

For a moderate size n , the naive algorithm is still worthwhile, but we can reduce some of the computation time by taking advantage of the polygons themselves. A polygon vertex V does not have to be compared to all other graph vertices. It suffices to compare it against those graph vertices lying in a *wedge* (or cone) determined by V and its two immediate neighbors, V_p and V_n . Figure 8.45 illustrates.

The *Wedge* class is a simple implementation for storing the triple of vertices and managing the point-in-wedge tests. The pseudocode for the graph construction is now

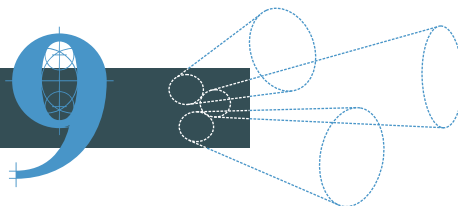
```

VisibilityGraph G;
G.Vertices = {waypoints, obstacle vertices, transporter vertices};
Wedges wedge = {compute wedges for each point in G.Vertices};
for each P in G.Vertices do
{
    for each Q in G.Vertices and in wedge[P] do
    {
        bool isGraphEdge = true;
        for each obstacle or transporter polygon edge E do
        {
            if segment (P,Q) intersects E at an interior point then
            {
                isGraphEdge = false;
                break;
            }
        }
    }
}

```

```
        if (isGraphEdge)
        {
            G.Edges.Insert(P,Q);
        }
    }
}
```

CHAPTER



PHYSICS

One of the most popular components in a game these days is the *physics engine*. This component encapsulates both collision detection and collision response. I talked about the details of a dynamic collision detection system in Chapter 8, but I have not talked much at all about collision response—the mechanism by which you translate, rotate, and possibly deform objects immediately after they have collided with other objects.

Physics itself is a very large topic, and the theory and implementation of physics is nontrivial. My goal in this book is to cover a wide range of topics regarding engines, so this chapter provides only a brief overview of some methods that are useful when incorporating physics into an application. These methods are not too difficult to implement. A full “black box” computational physics system that can handle arbitrary rigid bodies is a commercial venture these days. Iterative methods are employed for solving constrained optimizations that are inherent in rigid body physics. The mathematics behind these is quite advanced. For many more details, search online for recent research papers, and read my book [Ebe03].

The equations of motion for a physical simulation can always be written as a system of nonlinear equations of the form

$$\frac{d\mathbf{X}}{dt} = \mathbf{F}(t, \mathbf{X}), \quad t \geq 0, \quad \mathbf{X}(0) = \mathbf{X}_0 \quad (9.1)$$

The vector $\mathbf{X}$ represents the physical states of the simulation, typically including position, linear velocity, orientation, and angular velocity. The system is an initial-value problem since the state vector is specified at the initial time $t \geq 0$.

The differential equations are almost never solvable in closed form, so numerical methods must be used for approximating the solution. The simplest method is *Euler's method*. The idea is to replace the first derivative of Equation (9.1) by a forward difference approximation:

$$\frac{\mathbf{X}(t+h) - \mathbf{X}(t)}{h} = \mathbf{F}(t, \mathbf{X}(t))$$

The value $h > 0$ is the *step size* of the solver. Generally, the smaller the value of h , the less error you make in the approximation. This is solved for the term involving $t+h$:

$$\mathbf{X}(t+h) = \mathbf{X}(t) + h\mathbf{F}(t, \mathbf{X}(t)) \quad (9.2)$$

At time step t , if the state $\mathbf{X}(t)$ is known, then Euler's method gives you an approximation of the state at time $t+h$, namely, $\mathbf{X}(t+h)$.

Euler's method is the prototype for a numerical solver for ordinary differential equations. The function $\mathbf{F}$ is a given. Knowing the input time t , a step size h , and an input state $\mathbf{X}(t)$, the method produces an output time $t+h$ and a corresponding state $\mathbf{X}(t+h)$. The general concept is encapsulated by the abstract base class, `OdeSolver`, in the source code on the CD-ROM. Better methods than Euler's method are used in physics engines, especially when you need numerical stability. The most common of these methods is discussed briefly in Section 16.7. Much more detail on these methods and others is found in [Ebe03] or in standard textbooks on numerical methods (e.g., [BF01]).

9.1 PARTICLE SYSTEMS

The engine supports physical simulation of particle systems. The n particles are point sources with positions $\mathbf{X}_i$, masses m_i , and velocities $\mathbf{V}_i$, for $0 \leq i < n$. Forces $\mathbf{F}_i = m_i\mathbf{A}_i$ are applied to the particles, where $\mathbf{A}_i$ is the acceleration of the particle. The simulation is modeled using Newton's equations of motion:

$$\ddot{\mathbf{X}}_i = \mathbf{F}_i(t, \mathbf{X}_i, \dot{\mathbf{X}}_i)/m_i, \quad 0 \leq i < n$$

This is a second-order system of ordinary differential equations and is converted to a system of first-order equations:

$$\begin{bmatrix} \dot{\mathbf{X}}_i \\ \dot{\mathbf{V}}_i \end{bmatrix} = \begin{bmatrix} \mathbf{V}_i \\ \mathbf{F}_i(t, \mathbf{X}_i, \mathbf{V}_i)/m_i \end{bmatrix} \quad (9.3)$$

The system is solved using the Runge-Kutta fourth-order method.

The class that encapsulates the particle system is `ParticleSystem`. Its interface is listed next. The template parameters include `Real` for the floating-point type and `TVector`, which is either `Vector2` or `Vector3` to support 2D or 3D systems.

```

template <class Real, class TVector>
class ParticleSystem
{
public:
    ParticleSystem (int iNumParticles, Real fStep);
    virtual ~ParticleSystem ();

    int GetNumParticles () const;
    void SetMass (int i, Real fMass);
    Real GetMass (int i) const;
    TVector* Positions () const;
    TVector& Position (int i);
    TVector* Velocities () const;
    TVector& Velocity (int i);
    void SetStep (Real fStep);
    Real GetStep () const;

    virtual TVector Acceleration (int i, Real fTime,
        const TVector* akPosition, const TVector* akVelocity) = 0;

    virtual void Update (Real fTime);

protected:
    int m_iNumParticles;
    Real* m_afMass;
    Real* m_afInvMass;
    TVector* m_akPosition;
    TVector* m_akVelocity;
    Real m_fStep, m_fHalfStep, m_fSixthStep;

    // temporary storage for solver
    typedef TVector* TVectorPtr;
    TVectorPtr m_akPTmp, m_akDPTmp1, m_akDPTmp2;
    TVectorPtr m_akDPTmp3, m_akDPTmp4;
    TVectorPtr m_akVTmp, m_akDVTmp1, m_akDVTmp2;
    TVectorPtr m_akDVTmp3, m_akDVTmp4;
};

```

Many of the class member functions are accessors. The simulation is supported by the virtual functions `Acceleration` and `Update`. The right-hand side of Equation (9.3) has the force divided by mass, which is the acceleration of the particle. The member function `Acceleration` is what a derived class implements to represent $\mathbf{F}_i/m_i$ for each particle. The acceleration depends on the time t , the current particle position $\mathbf{X}_i$, and the current particle velocity $\mathbf{V}_i$.

The update function is the call into the Runge-Kutta solver. A particle is immovable if it has infinite mass. Equivalently, the inverse of the mass is zero. Only particles with finite mass are affected by the applied forces. You might compare the Runge-Kutta solver in `ParticleSystem` to the generic one in class `OdeRungeKutta4`. The particle system solver iterates over the particles, applying the Runge-Kutta algorithm to each one, but ignoring those particles with infinite mass. Notice that each step involves updating two arrays named `m_akDPTmp*` and `m_akDVTmp*`. The first type of array corresponds to $\dot{\mathbf{X}}_i = \mathbf{V}_i$ in Equation (9.3), and the second type of array corresponds to $\dot{\mathbf{V}}_i = \mathbf{F}_i(t, \mathbf{X}_i, \mathbf{V}_i)/m_i$ in Equation (9.3).

You may derive classes from `ParticleSystem` to build your own customized particle systems. The next section describes a few such classes that represent mass-spring systems.

9.2 MASS-SPRING SYSTEMS

A popular choice for modeling deformable bodies is mass-spring systems, which I discussed in detail in [Ebe03]. This section contains a brief summary of the ideas, of which two are important for implementation purposes. First, the springs connecting the masses are modeled using Hooke's law and lead to the equations of motion. I solve these numerically using Runge-Kutta fourth-order methods. Second, the topology of the connections of the masses by springs must be handled by an implementation. Curve masses are modeled as a 1D array of particles (e.g., hair or rope), surface masses as 2D arrays (e.g., cloth or water surface), and volume masses as 3D arrays (e.g., gelatinous blob or viscous material).

9.2.1 CURVE MASSES

A curve mass is represented as a polyline of vertices, open with two endpoints or closed with no endpoints. Each vertex of the polyline represents a mass. Each edge represents a spring connecting the two masses at the endpoints of the edge. Figure 9.1 shows two such configurations.

The equations of motion for an open linear chain are as follows. The masses m_i are located at positions $\mathbf{X}_i$ for $1 \leq i \leq p$. The system has $p - 1$ springs connecting the masses; spring i connects m_i and m_{i+1} . At an interior point i , two spring forces are applied, one from the spring shared with point $i - 1$ and one from the spring shared with point $i + 1$. The spring connecting masses m_i and m_{i+1} has spring constant c_i and rest length L_i . The differential equation for particle i is

$$m_i \ddot{\mathbf{x}}_i = c_{i-1} (|\mathbf{x}_{i-1} - \mathbf{x}_i| - L_{i-1}) \frac{\mathbf{x}_{i-1} - \mathbf{x}_i}{|\mathbf{x}_{i-1} - \mathbf{x}_i|} + c_i (|\mathbf{x}_{i+1} - \mathbf{x}_i| - L_i) \frac{\mathbf{x}_{i+1} - \mathbf{x}_i}{|\mathbf{x}_{i+1} - \mathbf{x}_i|} + \mathbf{F}_i \quad (9.4)$$

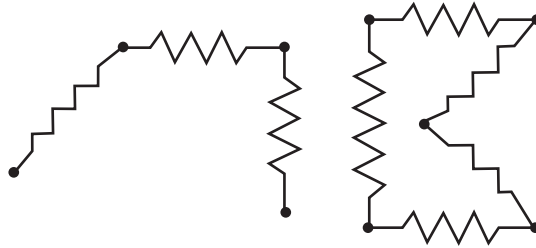


Figure 9.1 Two curve mass objects represented as mass-spring systems.

where $\mathbf{F}_i$ represents other forces acting on particle i , such as gravitational or wind forces. With the proper definitions at the two boundary particles of c_0 , c_p , L_0 , L_p , $\mathbf{X}_0$, and $\mathbf{X}_{p+1}$, Equation (9.4) also handles fixed boundary points and closed loops.

The class that implements a deformable curve mass is `MassSpringCurve` and is derived from `ParticleSystem`. The interface is

```
template <class Real, class TVector>
class MassSpringCurve : public ParticleSystem<Real,TVector>
{
public:
    MassSpringCurve (int iNumParticles, Real fStep);
    virtual ~MassSpringCurve ();

    int GetNumSprings () const;
    Real& Constant (int i); // spring constant
    Real& Length (int i); // spring resting length

    virtual TVector Acceleration (int i, Real fTime,
        const TVector* akPosition, const TVector* akVelocity);

    virtual TVector ExternalAcceleration (int i, Real fTime,
        const TVector* akPosition, const TVector* akVelocity);

protected:
    int m_iNumSprings;
    Real* m_afConstant;
    Real* m_afLength;
};
```

The number of particles in the mass-spring system is passed to the constructor. The second parameter, `fStep`, is the step size used in the Runge-Kutta numerical solver. After construction, you must set the spring constants and spring resting lengths via the appropriate member functions.

The `Acceleration` function is an override of the base class virtual function and is what is called by the Runge-Kutta numerical solver. This function handles the internal forces that the springs exert on the masses. The implementation is

```
template <class Real, class TVector>
TVector MassSpringCurve<Real,TVector>::Acceleration (int i, Real fTime,
    const TVector* akPosition, const TVector* akVelocity)
{
    TVector kAcceleration = ExternalAcceleration(i,fTime,
        akPosition,akVelocity);

    TVector kDiff, kForce;
    Real fRatio;

    if ( i > 0 )
    {
        int iM1 = i-1;
        kDiff = akPosition[iM1] - akPosition[i];
        fRatio = m_afLength[iM1]/kDiff.Length();
        kForce = m_afConstant[iM1]*(((Real)1.0)-fRatio)*kDiff;
        kAcceleration += m_afInvMass[i]*kForce;
    }

    int iP1 = i+1;
    if ( iP1 < m_iNumParticles )
    {
        kDiff = akPosition[iP1] - akPosition[i];
        fRatio = m_afLength[i]/kDiff.Length();
        kForce = m_afConstant[i]*(((Real)1.0)-fRatio)*kDiff;
        kAcceleration += m_afInvMass[i]*kForce;
    }

    return kAcceleration;
}
```

This is a straightforward implementation of the right-hand side of Equation (9.4). The endpoints of the curve of masses are handled separately since each has only one spring attached to it.

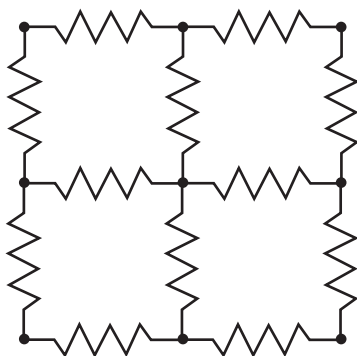


Figure 9.2 A surface mass represented as a mass-spring system with the masses organized as a 2D array.

We must also allow for external forces such as gravity, wind, and friction. The function `ExternalAcceleration` supports these. Just as in the `ParticleSystem` class, the function represents the acceleration $\mathbf{F}_i/m_i$ for a force $\mathbf{F}_i$ exerted on the particle i . Derived classes override this function, but the default implementation is for a zero external force.

The sample application on the CD-ROM that illustrates the use of `MassSpringCurve` is

`GeometricTools/WildMagic4/SamplePhysics/Rope`

The application models a rope as a deformable curve mass.

9.2.2 SURFACE MASSES

A surface mass is represented as a collection of particles arranged as a 2D array. An interior particle has four neighbors, as shown in Figure 9.2. The masses are m_{i_0, i_1} and are located at $\mathbf{X}_{i_0, i_1}$ for $0 \leq i_0 < n_0$ and $0 \leq i_1 < n_1$. The spring to the right of a particle has spring constant $c_{i_0, i_1}^{(0)}$ and resting length $L_{i_0, i_1}^{(0)}$. The spring below a particle has spring constant $c_{i_0, i_1}^{(1)}$ and resting length $L_{i_0, i_1}^{(1)}$. The understanding is that the spring constants and resting lengths are zero if the particle has no such spring in the specified direction.

The equation of motion for particle (i_0, i_1) has four force terms due to Hooke's law, one for each neighboring particle:

$$\begin{aligned}
m_{i_0, i_1} \ddot{\mathbf{X}}_{i_0, i_1} = & c_{i_0-1, i_1} \left(|\mathbf{X}_{i_0-1, i_1} - \mathbf{X}_{i_0, i_1}| - L_{i_0-1, i_1} \right) \frac{\mathbf{X}_{i_0-1, i_1} - \mathbf{X}_{i_0, i_1}}{|\mathbf{X}_{i_0-1, i_1} - \mathbf{X}_{i_0, i_1}|} \\
& + c_{i_0+1, i_1} \left(|\mathbf{X}_{i_0+1, i_1} - \mathbf{X}_{i_0, i_1}| - L_{i_0+1, i_1} \right) \frac{\mathbf{X}_{i_0+1, i_1} - \mathbf{X}_{i_0, i_1}}{|\mathbf{X}_{i_0+1, i_1} - \mathbf{X}_{i_0, i_1}|} \\
& + c_{i_0, i_1-1} \left(|\mathbf{X}_{i_0, i_1-1} - \mathbf{X}_{i_0, i_1}| - L_{i_0, i_1-1} \right) \frac{\mathbf{X}_{i_0, i_1-1} - \mathbf{X}_{i_0, i_1}}{|\mathbf{X}_{i_0, i_1-1} - \mathbf{X}_{i_0, i_1}|} \\
& + c_{i_0, i_1+1} \left(|\mathbf{X}_{i_0, i_1+1} - \mathbf{X}_{i_0, i_1}| - L_{i_0, i_1+1} \right) \frac{\mathbf{X}_{i_0, i_1+1} - \mathbf{X}_{i_0, i_1}}{|\mathbf{X}_{i_0, i_1+1} - \mathbf{X}_{i_0, i_1}|} \\
& + \mathbf{F}_{i_0, i_1}
\end{aligned} \tag{9.5}$$

As in the case of linear chains, with the proper definition of the spring constants and resting lengths at the boundary points of the mesh, Equation (9.5) applies to the boundary points as well as to the interior points.

The class that implements a deformable surface mass is `MassSpringSurface` and is derived from `ParticleSystem`. The interface is

```

template <class Real, class TVector>
class MassSpringSurface : public ParticleSystem<Real, TVector>
{
public:
    MassSpringSurface (int iRows, int iCols, Real fStep);
    virtual ~MassSpringSurface ();

    int GetRows () const;
    int GetCols () const;
    void SetMass (int iRow, int iCol, Real fMass);
    Real GetMass (int iRow, int iCol) const;
    TVector** Positions2D () const;
    TVector& Position (int iRow, int iCol);
    TVector** Velocities2D () const;
    TVector& Velocity (int iRow, int iCol);

    Real& ConstantR (int iRow, int iCol); // spring to (r+1,c)
    Real& LengthR (int iRow, int iCol); // spring to (r+1,c)
    Real& ConstantC (int iRow, int iCol); // spring to (r,c+1)
    Real& LengthC (int iRow, int iCol); // spring to (r,c+1)

    virtual TVector Acceleration (int i, Real fTime,
        const TVector* akPosition, const TVector* akVelocity);

```

```

virtual TVector ExternalAcceleration (int i, Real fTime,
    const TVector* akPosition, const TVector* akVelocity);

protected:
    int GetIndex (int iRow, int iCol) const;
    void GetCoordinates (int i, int& riRow, int& riCol) const;

    int m_iRows;           // R
    int m_iCols;           // C
    TVector** m_aakPosition; // R-by-C
    TVector** m_aakVelocity; // R-by-C

    int m_iRowsM1;        // R-1
    int m_iColsM1;        // C-1
    Real** m_aafConstantR; // (R-1)-by-C
    Real** m_aafLengthR;   // (R-1)-by-C
    Real** m_aafConstantC; // R-by-(C-1)
    Real** m_aafLengthC;   // R-by-(C-1)
};

```

This class represents an $R \times C$ array of masses lying on a surface and connected by an array of springs. The masses are indexed by $m_{r,c}$ for $0 \leq r < R$ and $0 \leq c < C$ and are stored in row-major order. The other arrays are also stored in linear memory in row-major order. The mass at interior position $\mathbf{X}_{r,c}$ is connected by springs to the masses at positions $\mathbf{X}_{r-1,c}$, $\mathbf{X}_{r+1,c}$, $\mathbf{X}_{r,c-1}$, and $\mathbf{X}_{r,c+1}$. Boundary masses have springs connecting them to the obvious neighbors: an “edge” mass has three neighbors and a “corner” mass has two neighbors.

The base class has support for accessing the masses, positions, and velocities stored in a linear array. Rather than force you to use the 1D index i for the 2D pair (r, c) , I have provided member functions for accessing the masses, positions, and velocities using the (r, c) pair. To avoid name conflict, `Positions` is used to access the 1D array of particles. In the derived class, `Positions2D` is the accessor for the same array, but as a 2D array. Simultaneous representations of the arrays require the class to use `System::Allocate` and `System::Deallocate` for dynamic creation and destruction of the array. The protected functions `GetIndex` and `GetCoordinates` implement the mapping between 1D and 2D indices.

The spring constants and spring resting lengths must be set after a class object is constructed. The interior mass at (r, c) has springs to the left, right, bottom, and top. Edge masses have only three neighbors, and corner masses have only two neighbors. The mass at (r, c) provides access to the springs connecting to locations $(r, c + 1)$ and $(r + 1, c)$. Edge and corner masses provide access to only a subset of these. The caller is responsible for ensuring the validity of the (r, c) inputs.

The virtual functions `Acceleration` and `ExternalAcceleration` are similar to the ones in class `MassSpringCurve`.

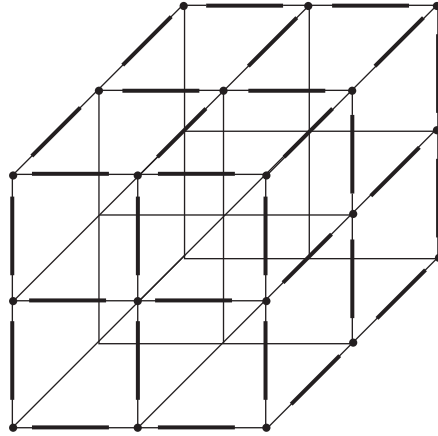


Figure 9.3 A volume mass represented as a mass-spring system with the masses organized as a 3D array. Only the masses and springs on the three visible faces are shown. The other connections are shown, but without their springs.

The sample application on the CD-ROM that illustrates the use of `MassSpringSurface` is

`GeometricTools/WildMagic4/SamplePhysics/Cloth`

The application models a cloth as a deformable surface mass.

9.2.3 VOLUME MASSES

A volume mass is represented as a collection of particles arranged as a 3D array. An interior particle has eight neighbors, as shown in Figure 9.3. The masses are m_{i_0, i_1, i_2} and are located at $\mathbf{X}_{i_0, i_1, i_2}$ for $0 \leq i_j < n_j$, $j = 0, 1, 2$. In the direction of positive increase of index i_j , the spring has a spring constant $c_{i_0, i_1, i_2}^{(j)}$ and resting length $L_{i_0, i_1, i_2}^{(j)}$ for $j = 0, 1, 2$. The understanding is that the spring constants and resting lengths are zero if the particle has no such spring in the specified direction.

The equation of motion for particle (i_0, i_1, i_2) has eight force terms due to Hooke's law, one for each neighboring particle:

$$\begin{aligned}
m_{i_0, i_1, i_2} \ddot{\mathbf{X}}_{i_0, i_1, i_2} = & \\
& + c_{i_0-1, i_1, i_2} \left(|\mathbf{X}_{i_0-1, i_1, i_2} - \mathbf{X}_{i_0, i_1, i_2}| - L_{i_0-1, i_1, i_2} \right) \frac{\mathbf{X}_{i_0-1, i_1, i_2} - \mathbf{X}_{i_0, i_1, i_2}}{|\mathbf{X}_{i_0-1, i_1, i_2} - \mathbf{X}_{i_0, i_1, i_2}|} \\
& + c_{i_0+1, i_1, i_2} \left(|\mathbf{X}_{i_0+1, i_1, i_2} - \mathbf{X}_{i_0, i_1, i_2}| - L_{i_0+1, i_1, i_2} \right) \frac{\mathbf{X}_{i_0+1, i_1, i_2} - \mathbf{X}_{i_0, i_1, i_2}}{|\mathbf{X}_{i_0+1, i_1, i_2} - \mathbf{X}_{i_0, i_1, i_2}|} \\
& + c_{i_0, i_1-1, i_2} \left(|\mathbf{X}_{i_0, i_1-1, i_2} - \mathbf{X}_{i_0, i_1, i_2}| - L_{i_0, i_1-1, i_2} \right) \frac{\mathbf{X}_{i_0, i_1-1, i_2} - \mathbf{X}_{i_0, i_1, i_2}}{|\mathbf{X}_{i_0, i_1-1, i_2} - \mathbf{X}_{i_0, i_1, i_2}|} \\
& + c_{i_0, i_1+1, i_2} \left(|\mathbf{X}_{i_0, i_1+1, i_2} - \mathbf{X}_{i_0, i_1, i_2}| - L_{i_0, i_1+1, i_2} \right) \frac{\mathbf{X}_{i_0, i_1+1, i_2} - \mathbf{X}_{i_0, i_1, i_2}}{|\mathbf{X}_{i_0, i_1+1, i_2} - \mathbf{X}_{i_0, i_1, i_2}|} \\
& + c_{i_0, i_1, i_2-1} \left(|\mathbf{X}_{i_0, i_1, i_2-1} - \mathbf{X}_{i_0, i_1, i_2}| - L_{i_0, i_1, i_2-1} \right) \frac{\mathbf{X}_{i_0, i_1, i_2-1} - \mathbf{X}_{i_0, i_1, i_2}}{|\mathbf{X}_{i_0, i_1, i_2-1} - \mathbf{X}_{i_0, i_1, i_2}|} \\
& + c_{i_0, i_1, i_2+1} \left(|\mathbf{X}_{i_0, i_1, i_2+1} - \mathbf{X}_{i_0, i_1, i_2}| - L_{i_0, i_1, i_2+1} \right) \frac{\mathbf{X}_{i_0, i_1, i_2+1} - \mathbf{X}_{i_0, i_1, i_2}}{|\mathbf{X}_{i_0, i_1, i_2+1} - \mathbf{X}_{i_0, i_1, i_2}|} \\
& + \mathbf{F}_{i_0, i_1, i_2}
\end{aligned} \tag{9.6}$$

With the proper definition of the spring constants and resting lengths at the boundary points of the mesh, Equation (9.6) applies to the boundary points as well as to the interior points.

The class that implements a deformable volume mass is `MassSpringVolume` and is derived from `ParticleSystem`. The interface is

```

template <class Real, class TVector>
class MassSpringVolume : public ParticleSystem<Real, TVector>
{
public:
    MassSpringVolume (int iSlices, int iRows, int iCols, Real fStep);
    virtual ~MassSpringVolume ();

    int GetSlices () const;
    int GetRows () const;
    int GetCols () const;
    void SetMass (int iSlice, int iRow, int iCol, Real fMass);
    Real GetMass (int iSlice, int iRow, int iCol) const;
    TVector*** Positions3D () const;
    TVector& Position (int iSlice, int iRow, int iCol);
    TVector*** Velocities3D () const;

```



```

TVector& Velocity (int iSlice, int iRow, int iCol);

Real& ConstantS (int iS, int iR, int iC); // to (s+1,r,c)
Real& LengthS (int iS, int iR, int iC);   // to (s+1,r,c)
Real& ConstantR (int iS, int iR, int iC); // to (s,r+1,c)
Real& LengthR (int iS, int iR, int iC);   // to (s,r+1,c)
Real& ConstantC (int iS, int iR, int iC); // to (s,r,c+1)
Real& LengthC (int iS, int iR, int iC);   // to (s,r,c+1)

virtual TVector Acceleration (int i, Real fTime,
    const TVector* akPosition, const TVector* akVelocity);

virtual TVector ExternalAcceleration (int i, Real fTime,
    const TVector* akPosition, const TVector* akVelocity);

protected:
int GetIndex (int iSlice, int iRow, int iCol) const;
void GetCoordinates (int i, int& riSlice, int& riRow,
    int& riCol) const;

int m_iSlices;           // S
int m_iRows;             // R
int m_iCols;             // C
int m_iSliceQuantity;    // R*C
TVector*** m_aaakPosition; // S-by-R-by-C
TVector*** m_aaakVelocity; // S-by-R-by-C

int m_iSlicesM1;         // S-1
int m_iRowsM1;           // R-1
int m_iColsM1;           // C-1
Real*** m_aaafConstants; // (S-1)-by-R-by-C
Real*** m_aaafLengthS;   // (S-1)-by-R-by-C
Real*** m_aaafConstantR; // S-by-(R-1)-by-C
Real*** m_aaafLengthR;   // S-by-(R-1)-by-C
Real*** m_aaafConstantC; // S-by-R-by-(C-1)
Real*** m_aaafLengthC;   // S-by-R-by-(C-1)
};

```

This class represents an $S \times R \times C$ array of masses lying in a volume and connected by an array of springs. The masses are indexed by $m(s, r, c)$ for $0 \leq s < S$, $0 \leq r < R$, and $0 \leq c < C$ and are stored in lexicographical order. That is, the index for the 1D array of memory is $i = c + C(r + Rs)$. The other arrays are also stored in linear memory in lexicographical order. The mass at interior position $\mathbf{X}_{s,r,c}$ is connected by springs to the masses at positions $\mathbf{X}_{s-1,r,c}$, $\mathbf{X}_{s+1,r,c}$, $\mathbf{X}_{s,r-1,c}$, $\mathbf{X}_{s,r+1,c}$, $\mathbf{X}_{s,r,c-1}$, and $\mathbf{X}_{s,r,c+1}$. Boundary masses have springs connecting them to the obvi-

ous neighbors: a “face” mass has five neighbors; an “edge” mass has four neighbors; and a “corner” mass has three neighbors.

The base class has support for accessing the masses, positions, and velocities stored in a linear array. Rather than force you to use the 1D index i for the 3D triple (s, r, c) , I have provided member functions for accessing the masses, positions, and velocities using the (s, r, c) triple. To avoid name conflict, `Positions` is used to access the 1D array of particles. In the derived class, `Positions3D` is the accessor for the same array, but as a 3D array. Simultaneous representations of the arrays require the class to use `System::Allocate` and `System::Deallocate` for dynamic creation and destruction of the array. The protected functions `GetIndex` and `GetCoordinates` implement the mapping between 1D and 3D indices.

The spring constants and spring resting lengths must be set after a class object is constructed. The interior mass at (s, r, c) has springs attaching it to six neighbors. Face masses have only five neighbors; edge masses have only four neighbors; and corner masses have only three neighbors. The mass at (s, r, c) provides access to the springs connecting to locations $(s + 1, r, c)$, $(s, r + 1, c)$, and $(s, r, c + 1)$. Face, edge, and corner masses provide access to only a subset of these. The caller is responsible for ensuring the validity of the (s, r, c) inputs.

The virtual functions `Acceleration` and `ExternalAcceleration` are similar to the ones in classes `MassSpringCurve` and `MassSpringSurface`.

The sample application on the CD-ROM that illustrates the use of `MassSpringVolume` is

`GeometricTools/WildMagic4/SamplePhysics/GelatinCube`

The application models a gelatinous cube as a deformable volume mass.

9.2.4 ARBITRARY CONFIGURATIONS

In general you can set up an arbitrary configuration for a mass-spring system of p particles with masses m_i and location $\mathbf{x}_i$. Each spring added to the system connects two masses, say, m_i and m_j . The spring constant is $c_{ij} > 0$, and the resting length is L_{ij} .

Let $\mathcal{A}_i$ denote the set of indices j such that m_j is connected to m_i by a spring—the set of *adjacent indices*, so to speak. The equation of motion for particle i is

$$m_i \ddot{\mathbf{x}}_i = \sum_{j \in \mathcal{A}_i} c_{ij} (|\mathbf{x}_j - \mathbf{x}_i| - L_{ij}) \frac{\mathbf{x}_j - \mathbf{x}_i}{|\mathbf{x}_j - \mathbf{x}_i|} + \mathbf{F}_i \quad (9.7)$$

The technical difficulty in building a differential equation solver for an arbitrary graph is encapsulated solely by a vertex-edge table that stores the graph. Whenever the numerical solver must process particle i via Equation (9.7), it must be able to iterate over the adjacent indices to evaluate the Hooke’s law terms.

The class that implements a mass-spring system with an arbitrary configuration of masses and springs is `MassSpringArbitrary` and is derived from `ParticleSystem`. The interface is

```
template <class Real, class TVector>
class MassSpringArbitrary : public ParticleSystem<Real,TVector>
{
public:
    MassSpringArbitrary (int iNumParticles, int iNumSprings,
        Real fStep);
    virtual ~MassSpringArbitrary ();

    int GetNumSprings () const;
    void SetSpring (int iSpring, int iParticle0, int iParticle1,
        Real fConstant, Real fLength);
    void GetSpring (int iSpring, int& riParticle0,
        int& riParticle1, Real& rfConstant, Real& rfLength) const;

    Real& Constant (int iSpring);
    Real& Length (int iSpring);

    virtual TVector Acceleration (int i, Real fTime,
        const TVector* akPosition, const TVector* akVelocity);

    virtual TVector ExternalAcceleration (int i, Real fTime,
        const TVector* akPosition, const TVector* akVelocity);

protected:
    class Spring
    {
    public:
        int Particle0, Particle1;
        Real Constant, Length;
    };

    int m_iNumSprings;
    Spring* m_akSpring;

    // Each particle has an associated array of spring indices for those
    // springs adjacent to the particle. The set elements are spring
    // indices, not indices of adjacent particles.
    TSet<int>* m_akAdjacent;
};
```

The constructor requires you to specify the number of particles and the number of springs in the system. The parameter `fStep` is the step size used in the Runge-Kutta numerical solver. After construction, you must call `SetSpring` for each spring that you want in the system. If spring i connects particles p_1 and p_2 , the order of the parameters in the function call is irrelevant. It is possible to have two springs that connect the same pair of particles, but I suggest using at most one spring per pair.

The array member, `m_akAdjacent`, is an array of sets of integers. The set `m_akAdjacent[i]` represents $\mathcal{A}_i$ and contains those integers j for which a spring connects particle i to particle j .

The virtual functions `Acceleration` and `ExternalAcceleration` are similar to the ones in classes `MassSpringCurve`, `MassSpringSurface`, and `MassSpringVolume`.

The sample application on the CD-ROM that illustrates the use of `MassSpringArbitrary` is

`GeometricTools/WildMagic4/SamplePhysics/GelatinBlob`

The application models a gelatinous blob as a deformable volume mass. The blob has the topology of an icosahedron.

9.3 DEFORMABLE BODIES

There are many ways to model deformable bodies in a physics simulation. A model that is designed to conform to the physical principles of deformation will most likely be expensive to compute in a real-time application. Instead, you should consider less expensive alternatives. I will mention a few possibilities here.

The `SurfaceMesh` class supports dynamic updating of the mesh vertices, which makes it a good candidate for representing a deformable body. You must provide the physics simulation that modifies the mesh vertices during run time. A pitfall of simulations is allowing arbitrary motion of vertices, which leads to self-intersections of the mesh. A collision detection system can help you determine—and prevent—self-intersections, but by doing so, you add an additional layer of expense to the computations. Once physics hardware becomes available on consumer machines, the expense will be negligible.

The `MorphController` class similarly allows you to dynamically deform a mesh. Whereas the deformations of `SurfaceMesh`-derived objects are controlled by changing the surface parameters of the derived class, the deformations of objects with a `MorphController` attached are controlled by a set of keyframes. The keyframes themselves may be dynamically modified.

Yet more classes in the engine that support deformable objects are `PointController` and `ParticleController`. Their interfaces allow you to specify the positions and velocities whenever you choose. A physical simulation will set these quantities accordingly.

The possibilities are endless. The `IKController` and `SkinController` classes may also be used for deformation. Animation via controllers directly supports the concept of deformation: It is just a matter of animating the data that directly, or indirectly, affects the vertices of a mesh. The popular rag doll physics is an excellent example of how to blend together deformable objects and collision detection and response.

9.4 RIGID BODIES

The last topic of the chapter is *rigid bodies*. Creating a general physics engine that handles interacting bodies is quite difficult. However, an engine will contain the foundations for computing unconstrained motion using Newton's equations of motion. The collision detection system computes the physical constraints that occur during run time. A careful separation of the collision detection subsystem and the collision response subsystem is called for. The two subsystems must interact, but the separation allows you to more easily diagnose problems and identify which subsystem is causing problems when your simulation shows that some objects are not conforming to the physical principles you had in mind.

A particle can be thought of as a rigid body without size or orientation; it has a position, velocity, and applied forces. Many objects in a physical simulation, though, are not particles, yet have size and orientation. The standard representation for a rigid body in a real-time application is a polyhedron. A coordinate system is chosen for the body for the purposes of positioning and orienting the object in space. For physical and mathematical reasons, the center of mass is chosen to be the body origin, and the body coordinate axes are chosen to be the principal directions of inertia. The direction vectors turn out to be eigenvectors of the inertia tensor for the body. The choice of coordinate system allows us to decompose the motion calculations into translation of the center of mass (position, linear velocity, and linear acceleration) and rotation of the body (orientation, angular velocity, and angular acceleration). Yet another simplifying assumption is that the mass of the rigid body is uniformly distributed within the body.

Here is a very brief summary of the material in [Ebe03] regarding unconstrained motion of a rigid body. Let $\mathbf{X}(t)$ and $\mathbf{V}(t)$ denote the position and velocity, respectively, of the center of mass of the rigid body. The linear momentum of the body is

$$\mathbf{P}(t) = m\mathbf{V}(t) \quad (9.8)$$

Newton's second law of motion states that the rate of change of linear momentum is equal to the applied force,

$$\dot{\mathbf{P}}(t) = \mathbf{F}(t) \quad (9.9)$$

where m is the mass of the body and $\mathbf{F}(t)$ is the applied force on the object. The equations of motion pertaining to position and linear momentum are

$$\dot{\mathbf{X}} = m^{-1}\mathbf{P}, \quad \dot{\mathbf{P}} = \mathbf{F} \quad (9.10)$$

Similar equations of motion can be derived for the orientation matrix $R(t)$ of the body. In the coordinate system of the rigid body, let $\mathbf{b}$ denote the time-independent position of a point relative to the origin (the center of mass). The world coordinate of the point is

$$\mathbf{Y}(t) = \mathbf{X}(t) + R(t)\mathbf{b}$$

The inertia tensor in body coordinates is the 3×3 symmetric matrix,

$$J_{\text{body}} = \int_B \left(|\mathbf{b}|^2 I - \mathbf{b}\mathbf{b}^T \right) dm \quad (9.11)$$

where B is the set of points making up the body, I is the identity matrix, and dm is the infinitesimal measure of mass in the body. For a body of constant density δ , $dm = \delta dV$, where dV is the infinitesimal measure of volume in the body. As we will see later in this section, the integration in Equation (9.11) can be computed exactly for a constant-density, rigid body that is represented by a polyhedron. The resulting formula is an algebraic expression that is easily computed.

The inertia tensor in world coordinates is

$$J(t) = \int_B \left(|\mathbf{r}|^2 I - \mathbf{r}\mathbf{r}^T \right) dm = R(t)J_{\text{body}}R(t)^T \quad (9.12)$$

where $\mathbf{r}(t) = \mathbf{Y}(t) - \mathbf{X}(t) = R(t)\mathbf{b}$. The inertia tensor is sometimes referred to as the *mass matrix*.

The rate of change of the orientation matrix, $R(t)$, is related to the angular velocity vector, $\mathbf{W}(t) = (w_0, w_1, w_2)$, by

$$\dot{R}(t) = \text{Skew}(\mathbf{W}(t))R(t) \quad (9.13)$$

where $S = \text{Skew}(\mathbf{W})$ is the skew-symmetric matrix whose entries are $S_{00} = S_{11} = S_{22} = 0$, $S_{01} = -w_2$, $S_{10} = w_2$, $S_{02} = w_1$, $S_{20} = -w_1$, $S_{12} = -w_0$, and $S_{21} = w_0$.

The angular momentum, $\mathbf{L}(t)$, and angular velocity, $\mathbf{W}(t)$, are related by

$$\mathbf{L}(t) = J(t)\mathbf{W}(t) \quad (9.14)$$

where $J(t)$ is the inertia tensor defined by Equation (9.12). Notice the similarity of Equation (9.14) to Equation (9.8). The linear momentum is defined as *mass times linear velocity*, and the angular momentum is defined as *mass matrix times angular velocity*.

The equivalent of Newton's second law of motion, which relates linear acceleration and force, is the following, which states that the rate of change of angular momentum is equal to the applied torque:

$$\dot{\mathbf{L}}(t) = \boldsymbol{\tau}(t) \quad (9.15)$$

where $\boldsymbol{\tau}(t)$ is the torque applied to the rigid body.

The equations of motion pertaining to orientation and angular momentum are

$$\dot{\mathbf{R}} = \text{Skew}(\mathbf{W})\mathbf{R}, \quad \dot{\mathbf{L}} = \boldsymbol{\tau} \quad (9.16)$$

The angular velocity is dependent on other known quantities, namely,

$$\mathbf{W}(t) = \mathbf{J}^{-1}\dot{\mathbf{L}} = \mathbf{R}\mathbf{J}_{\text{body}}^{-1}\mathbf{R}^T\dot{\mathbf{L}} \quad (9.17)$$

Equations (9.10), (9.16), and (9.17) can be combined into a single system of differential equations that model the unconstrained motion of the rigid body. The state vector is $\mathbf{S} = (\mathbf{X}, \mathbf{P}, \mathbf{R}, \mathbf{L})$, and the system of equations is

$$\frac{d\mathbf{S}}{dt} = \frac{d}{dt} \begin{bmatrix} \mathbf{X} \\ \mathbf{R} \\ \mathbf{P} \\ \mathbf{L} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{X}} \\ \dot{\mathbf{R}} \\ \dot{\mathbf{P}} \\ \dot{\mathbf{L}} \end{bmatrix} = \begin{bmatrix} m^{-1}\mathbf{P} \\ \text{Skew}(\mathbf{R}\mathbf{J}_{\text{body}}^{-1}\mathbf{R}^T\dot{\mathbf{L}})\mathbf{R} \\ \mathbf{F} \\ \boldsymbol{\tau} \end{bmatrix} = \mathbf{G}(t, \mathbf{S}) \quad (9.18)$$

This system is first-order, so it may be solved numerically using your favorite differential equation solver. My choice is the Runge-Kutta fourth-order method. The input parameters are the mass m and body inertia tensor $\mathbf{J}_{\text{body}}$, both constants during the physical simulation. The force $\mathbf{F}$ and torque $\boldsymbol{\tau}$ are vector-valued functions that your application must present to the simulator. The initial state, $\mathbf{S}(0)$, is also specified by your application. Once all these quantities are known, the numerical solver is ready to be iterated.

Although Equation (9.18) is ready to solve numerically, most practitioners choose to use quaternions to represent the orientation matrices. If $\mathbf{R}(t)$ is the orientation matrix, a corresponding quaternion is denoted $q(t)$. The equivalent of Equation (9.13) for quaternions is

$$\dot{q}(t) = \omega(t)q(t)/2 \quad (9.19)$$

where $\omega = W_0i + W_1j + W_2k$ is the quaternion representation of the angular velocity $\mathbf{W} = (W_0, W_1, W_2)$. The system of equations that I really implement is

$$\begin{bmatrix} \dot{\mathbf{X}} \\ \dot{q} \\ \dot{\mathbf{P}} \\ \dot{\mathbf{L}} \end{bmatrix} = \begin{bmatrix} m^{-1}\mathbf{P} \\ \omega q/2 \\ \mathbf{F} \\ \boldsymbol{\tau} \end{bmatrix} \quad (9.20)$$

After each iteration of the numerical solver, the application must transform the rigid body to its new world coordinates. The use of quaternions will require us to convert between quaternions and rotation matrices. The classes `Matrix3` and `Quaternion` support these conversions.

9.4.1 THE RIGID BODY CLASS

The class that encapsulates a rigid body is `RigidBody` and has the interface

```
template <class Real>
class RigidBody
{
public:
    RigidBody ();
    virtual ~RigidBody ();

    // Set/Get position.
    Vector3<Real>& Position ();

    // Set rigid body state.
    void SetMass (float fMass);
    void SetBodyInertia (const Matrix3<Real>& rkInertia);
    void SetPosition (const Vector3<Real>& rkPos);
    void SetQOrientation (const Quaternion<Real>& rkQOrient);
    void SetLinearMomentum (const Vector3<Real>& rkLinMom);
    void SetAngularMomentum (const Vector3<Real>& rkAngMom);
    void SetROrientation (const Matrix3<Real>& rkROrient);
    void SetLinearVelocity (const Vector3<Real>& rkLinVel);
    void SetAngularVelocity (const Vector3<Real>& rkAngVel);

    // Get rigid body state.
    Real GetMass () const;
    Real GetInverseMass () const;
    const Matrix3<Real>& GetBodyInertia () const;
    const Matrix3<Real>& GetBodyInverseInertia () const;
    Matrix3<Real> GetWorldInertia () const;
    Matrix3<Real> GetWorldInverseInertia () const;
    const Vector3<Real>& GetPosition () const;
    const Quaternion<Real>& GetQOrientation () const;
    const Vector3<Real>& GetLinearMomentum () const;
    const Vector3<Real>& GetAngularMomentum () const;
    const Matrix3<Real>& GetROrientation () const;
    const Vector3<Real>& GetLinearVelocity () const;
    const Vector3<Real>& GetAngularVelocity () const;
```



```

// Force/Torque function format.
typedef Vector3<Real> (*Function)
(
    Real,                // time of application
    Real,                // mass
    const Vector3<Real>&, // position
    const Quaternion<Real>&, // orientation
    const Vector3<Real>&, // linear momentum
    const Vector3<Real>&, // angular momentum
    const Matrix3<Real>&, // orientation
    const Vector3<Real>&, // linear velocity
    const Vector3<Real>&  // angular velocity
);

// force and torque functions
Function Force;
Function Torque;

// Runge-Kutta fourth-order differential equation solver
void Update (Real fT, Real fDT);

protected:
    // constant quantities (matrices in body coordinates)
    Real m_fMass, m_fInvMass;
    Matrix3<Real> m_kInertia, m_kInvInertia;

    // state variables
    Vector3<Real> m_kPos;           // position
    Quaternion<Real> m_kQOrient;    // orientation
    Vector3<Real> m_kLinMom;        // linear momentum
    Vector3<Real> m_kAngMom;        // angular momentum

    // derived state variables
    Matrix3<Real> m_kROrient;       // orientation matrix
    Vector3<Real> m_kLinVel;        // linear velocity
    Vector3<Real> m_kAngVel;        // angular velocity
};

```

The constructor creates an uninitialized rigid body. The rigid body state must be initialized using the Set functions before starting the physical simulation. The Get functions allow you access to the current state of the rigid body.

The constant quantities for the rigid body are the mass and inertia tensor in body coordinates. Because the differential equation solver must divide by mass and use the inverse of the inertia tensor, these are computed once and stored. If you want a rigid

body to be immovable, set its *inverse* mass to zero and its *inverse* inertia tensor to the zero matrix. In effect, the body mass is infinite, and the body is too heavy to rotate.

The state variable in Equation (9.20) includes position, orientation (represented as a quaternion), linear momentum, and angular momentum. These values are stored by the class. The other quantities of interest are derived from the state variables: the orientation matrix (derived from the quaternion orientation), the linear velocity (derived from the linear momentum and mass), and the angular velocity (derived from the angular momentum, the inertia tensor, and the orientation matrix). The derived variables are guaranteed to be synchronized with the state variables.

The class defines a function type, called `RigidBody::Function`. The force $\mathbf{F}$ and torque $\boldsymbol{\tau}$ in Equation (9.20) possibly depend on many variables, including the current time and state of the system. If you think of the equations of motion as $\dot{\mathbf{S}} = \mathbf{G}(t, \mathbf{S})$, then the function type `RigidBody::Function` represents the function on the right-hand side, $\mathbf{G}(t, \mathbf{S})$. The class has two data members that are in public scope, `Force` and `Torque`, which are set by your application.

The member function `Update` is a single iteration of the Runge-Kutta fourth-order solver. Its structure is similar to the previous implementations we have seen for the Runge-Kutta solvers. The exception is that after each of the four steps in the solver, the derived variables must be computed. The orientation matrix is computed from the orientation quaternion, the linear velocity is computed from the linear momentum and inverse mass, and the angular velocity is computed from the orientation matrix, the inverse inertia tensor, and the angular momentum.

Two applications illustrating the use of `RigidBody` are on the CD-ROM:

```
MagicSoftware/WildMagic4/Test/TestBouncingBalls
MagicSoftware/WildMagic4/Test/TestBouncingTetrahedra
```

The first application is fairly simple from the point of view of collision detection—it is easy to compute the contact time and contact point between two spheres. The second application is more complicated: it sets up the collision detection as a linear complementarity problem (LCP) and uses a numerical solver for the LCP. LCPs and their numerical solution are a complicated topic that I will not discuss here. See [Ebe03] for details and references to the literature.

9.4.2 COMPUTING THE INERTIA TENSOR

The `RigidBody` class requires you to initialize the body by specifying its mass and body inertia tensor. Generally, the inertia tensor is a complicated, mathematical beast. For constant-density bodies that are represented by polyhedra, the tensor can be computed in closed form. An efficient algorithm is in [Ebe03]. The paper [Mir96] is what practitioners had been using for the equations but is less efficient regarding the calculations, and the equations are more detailed and tedious to implement. The algorithm in [Ebe03] requires triangle faces for the polyhedra. A small extension of

the algorithm was made in [Kal] that allows simple polygon faces, but no details are presented here since the author of the paper and the company he works for chose to file a patent application.

The essence of the algorithm is that the entries of the inertia tensor are triple integrals evaluated over the region of space occupied by the body. Each integral has an integrand that is a quadratic polynomial. The divergence theorem from calculus allows you to convert the volume integrals to surface integrals. Because the object is a polyhedron, the surface integrals are reduced to a sum of integrals over the polyhedron faces. Each of these integrals is easily computed in closed form.

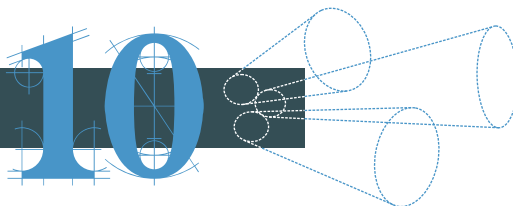
A single function is provided for computing the mass, center of mass, and the inertia tensor for a rigid body with constant density and represented by a polyhedron:

```
template <class Real>
void ComputeMassProperties (const Vector3<Real>* akVertex,
    int iTQuantity, const int* aiIndex, bool bBodyCoords,
    Real& rfMass, Vector3<Real>& rkCenter,
    Matrix3<Real>& rkInertia);
```

The polyhedron must be represented by a closed triangle mesh. Each edge of the mesh is shared by exactly two triangles. The first parameter of the function is the array of vertices for the mesh. The second parameter is the number of triangles in the mesh. The third parameter is the index array, which has $3T$ indices for T triangles. Each triple of indices represents a triangle in the mesh, and the indices are for lookups in the vertex array.

The parameter `bBodyCoords` is set to `true` when you want the inertia tensor in body coordinates. For the purpose of the class `RigidBody`, this is what you want. If you want the inertia tensor in world coordinates, set the Boolean parameter to `false`.

The last three parameters (the mass of the body, the center of mass, and the inertia tensor) are the output of the function.



STANDARD OBJECTS

The objects described here are common in 3D applications. It is useful to know how you can represent the objects to manipulate them in an engine or application.

10.1 LINEAR COMPONENTS

Points and vectors, two terms that are not identical in meaning (see Section 2.1.3), are simple enough to represent and manipulate in a computer program. Objects that are just as familiar to you are *linear components*, a general term that refers to *lines*, *rays*, and *line segments*. I will use the term *segment* rather than carrying around the modifier “line.” Linear components are useful in picking operations for object selection. They are also useful for stabbing operations, where you cast a few rays from the observer to see what objects are close (or far) to get an idea if the observer can move in some direction without colliding with something.

A *line* is defined parametrically by $\mathbf{X}(t) = \mathbf{P} + t\mathbf{D}$, where $\mathbf{P}$ is a point called the *origin* of the line, $\mathbf{D}$ is a vector called the *direction* of the line, and t is any real number. My convention is that a direction vector always has unit length. Sometimes you will see the interval notation used, $t \in (-\infty, \infty)$, where the interval shown is the set of all real numbers.

A *ray* is defined parametrically by $\mathbf{X}(t) = \mathbf{P} + t\mathbf{D}$, but the restriction on the parameter is $t \geq 0$, sometimes written using interval notation as $t \in [0, \infty)$.

A *segment* is typically defined by two endpoints $\mathbf{P}_0$ and $\mathbf{P}_1$. A parametric representation is $\mathbf{X}(t) = (1 - t)\mathbf{P}_0 + t\mathbf{P}_1$ for $t \in [0, 1]$. Equivalently, $\mathbf{X}(t) = \mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0)$ for $t \in [0, 1]$. A vector in the direction of the segment is $\mathbf{E} = \mathbf{P}_1 - \mathbf{P}_0$, but it is not usually unit length. The unit-length vector is $\mathbf{D} = \mathbf{E}/|\mathbf{E}|$.

An alternate representation is one that is consistent with the line and ray representations. Let $\mathbf{C}$ be the center (midpoint) of the segment and let $\mathbf{D}$ be a direction vector for the segment (there are two). Let the segment have radius (half-length, extent) r . The segment is defined parametrically by $\mathbf{X}(t) = \mathbf{P} + t\mathbf{D}$ for $|t| \leq r$.

Why would you choose one representation over the other? Consider the following argument. When attempting to determine the intersection of a line segment and a planar object such as a plane, triangle, or rectangle, you need to determine if the line segment is not parallel to the plane of the object. If it is not, there is a possibility that the segment and object intersect, but of course more work must be done to compute an intersection point. The work depends on the type of object you are dealing with. If the segment is parallel to the plane of the object, it is either separated from the plane or in the plane itself. In the latter case, there is a chance that the segment intersects the object, but now the problem is one that is really two-dimensional. Any intersection algorithm for segments must, therefore, classify whether the segment and plane are nonparallel, parallel and separated, or parallel and coplanar.

The classification involves testing if the segment direction $\mathbf{D}$ and the unit-length plane normal $\mathbf{N}$ are perpendicular. You compute $d = \mathbf{N} \cdot \mathbf{D}$. If $d \neq 0$, the segment is not parallel to the plane. If $d = 0$, the segment is parallel, in which case you need only test if a segment endpoint is on the plane (coplanar objects) or off the plane (separated objects).

Mathematically, the classification is straightforward to do. Numerically, though, you can run into problems. First, let us look more closely at the parallel case when we need to know if d is zero. This pseudocode is a disaster waiting to happen:

```
float d = Dot(plane.N,segment.D);
if (d != 0.0f)
{
    // segment not parallel to plane
}
else
{
    // segment parallel to plane
}
```

The theoretical value of d is what you want to be zero, but numerical round-off errors in computing d have really given you a floating-point number d' , which hopefully is nearly zero. For randomly generated segment directions and plane normals, the probability that you will reach the code block for the parallel case is effectively zero. Instead, your pseudocode should be

```
const float epsilon = <a small positive error tolerance>;
float d = Dot(plane.N,segment.D);
if (fabs(d) > epsilon)
```

```

{
    // segment not parallel to plane
}
else
{
    // segment parallel (or nearly parallel) to plane
}

```

Look at this in terms of angle θ between $\mathbf{N}$ and $\mathbf{D}$. We know that $\cos \theta = \mathbf{N} \cdot \mathbf{D}$. When $d = 0$, the angle is $\theta = \pi/2$. The code says that the segment is parallel to the plane when

$$|\cos \theta| \leq \varepsilon$$

for a small positive threshold ε . This is equivalent to

$$|\theta - \pi/2| \leq \phi$$

where $\phi > 0$ is chosen such that $\cos(\pi/2 - \phi) = \varepsilon$. For example, if $\varepsilon = 1\text{e-}06$, then $\phi \doteq 1\text{e-}06$ radians. This is reasonable. For this error tolerance, theoretically nonparallel segments and planes will be classified as parallel only when the angle between the segment direction and the plane normal is no more than $1\text{e-}06$ different from $\pi/2$.

Now suppose you used the difference of endpoints, $\mathbf{E}$, instead of a unit-length direction $\mathbf{D}$. The dot product is now $d = \mathbf{N} \cdot \mathbf{E}$ and you still compare d to zero. For the segment and plane to be parallel, you need $d = 0$. Using the pseudocode with an error tolerance:

```

const float epsilon = <a small positive error tolerance>;
Vector E = segment.P1 - segment.P0;
float d = Dot(plane.N,E);
if (fabs(d) > epsilon)
{
    // segment not parallel to plane
}
else
{
    // segment parallel (or nearly parallel) to plane
}

```

The numerical classification to be parallel *is not the same as before*. What we really have here is

$$|d| = |\mathbf{E}| |\cos \theta| \leq \varepsilon$$

where $|\mathbf{E}|$ is the length of $\mathbf{E}$. To make a comparison with the previous example, suppose $\varepsilon = 1\text{e-}06$. Two problems can occur in the current example. First, suppose that the endpoints are very close together, say, separated by a distance of $1\text{e-}06$. The test on the absolute dot product becomes

$$|\cos \theta| \leq 1$$

because $|\mathbf{E}|$ and ε are the same value and can be canceled in the expression. The angle condition is satisfied for all angles! No matter how the segment is oriented in space, it will be classified as parallel to the plane. Second, suppose that the endpoints are very far apart, say, separated by a distance of $1\text{e}+06$. The test on the absolute dot product becomes

$$|\cos \theta| \leq 1\text{e-}12$$

For 32-bit, floating-point numbers, quantities such as $1\text{e-}12$ are in the noise range. The probability of satisfying the condition is effectively zero, in which case no matter how the segment is oriented in space, it will never be classified as parallel to the plane.

I once always used the two-point representation for line segments. After constantly dealing with the floating-point problems, I switched to the center-direction-radius representation. The outcome has been good, but users of my code base sometimes miss the fact that I have a representation different from what they were expecting.

Ah, well, floating-point arithmetic is the curse of computational geometry.

10.2 PLANAR COMPONENTS

A *plane* in 3D space may be represented in a few ways. The first representation is to use three noncollinear points $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$. This representation is not that convenient for graphics systems. Instead, let $\mathbf{P}$ be a point on the plane, call it the *plane origin*, and let $\mathbf{N}$ be a unit-length normal to the plane. A point $\mathbf{X}$ on the plane satisfies the condition $\mathbf{N} \cdot (\mathbf{X} - \mathbf{P}) = 0$. All this says is that the vector $\mathbf{X} - \mathbf{P}$ is perpendicular to the normal. Alternatively, you may write $\mathbf{N} \cdot \mathbf{X} = d$, where d is referred to as a *plane constant*. No mention is made in the formula about a point on the plane, but if you need one, you can use $\mathbf{P} = d\mathbf{N}$. In fact, $\mathbf{P}$ is the point on the plane that is closest to the origin of the space containing the plane. The distance from the plane to the origin is $|d|$.

Sometimes you need a coordinate system that includes the plane origin $\mathbf{P}$ and the plane normal $\mathbf{N}$. You may choose two vectors $\mathbf{U}$ and $\mathbf{V}$ so that $\{\mathbf{U}, \mathbf{V}, \mathbf{N}\}$ is a right-handed orthonormal set (see Section 2.1.1). Any point $\mathbf{X}$ in space is written in this coordinate system as

$$\mathbf{X} = \mathbf{P} + y_0\mathbf{U} + y_1\mathbf{V} + y_2\mathbf{N} = \mathbf{P} + \mathbf{R}\mathbf{Y} \quad (10.1)$$

where R is the rotation matrix whose columns are $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{N}$ (in that order) and $\mathbf{Y} = (y_0, y_1, y_2)$ but is thought of as a column vector for the purpose of multiplication with R . One robust algorithm for choosing $\mathbf{U}$ and $\mathbf{V}$ is

```
Vector3 N = <unit-length plane normal>;
Vector3 U, V;
float invLength;
if (|N.x| >= |N.y|)
{
    // N.x or N.z is the largest-magnitude component; swap them.
    invLength = 1/sqrt(N.x*N.x + N.z*N.z);
    U.x = -N.z*invLength;
    U.y = 0;
    U.z = +N.x*invLength;

    // V = Cross(N,U)
    V.x = N.y*U.z;
    V.y = N.z*U.x - N.x*U.z;
    V.z = -N.y*U.x;
}
else
{
    // N.y or N.z is the largest-magnitude component; swap them.
    invLength = 1/sqrt(N.y*N.y + N.z*N.z);
    U.x = 0;
    U.y = +N.z*invLength;
    U.z = -N.y*invLength;

    // V = Cross(N,U)
    V.x = N.y*U.z - N.z*U.y;
    V.y = -N.x*U.z;
    V.z = N.x*U.y;
}
```

By *planar component*, I mean any 2D object living in a 3D space. Naturally, there are more planar components than you can count, but the two of interest here are triangles and rectangles.

A triangle consists of three noncollinear points (vertices) $\mathbf{P}_i$ for $0 \leq i \leq 2$. The plane that contains the triangle may be chosen so that $\mathbf{P}_0$ is the plane origin and the plane normal is

$$\mathbf{N} = \frac{(\mathbf{P}_1 - \mathbf{P}_0) \times (\mathbf{P}_2 - \mathbf{P}_0)}{|(\mathbf{P}_1 - \mathbf{P}_0) \times (\mathbf{P}_2 - \mathbf{P}_0)|}$$

Any point on the triangle may be represented in plane coordinates using Equation (10.1). More common, though, is to write the points using *barycentric coordinates*:

$$\mathbf{X} = b_0\mathbf{P}_0 + b_1\mathbf{P}_1 + b_2\mathbf{P}_2$$

where $b_i \in [0, 1]$ for all i and $b_0 + b_1 + b_2 = 1$. For any point in the plane of the triangle, the same representation works but the b_i may be any real numbers. Given $\mathbf{X}$, you may solve for the barycentric coordinates:

$$\mathbf{X} - \mathbf{P}_0 = (b_0 - 1)\mathbf{P}_0 + b_1\mathbf{P}_1 + b_2\mathbf{P}_2 = b_1(\mathbf{P}_1 - \mathbf{P}_0) + b_2(\mathbf{P}_2 - \mathbf{P}_0)$$

Applying various dot products, you may set up the linear system of equations

$$\begin{bmatrix} (\mathbf{P}_1 - \mathbf{P}_0) \cdot (\mathbf{P}_1 - \mathbf{P}_0) & (\mathbf{P}_1 - \mathbf{P}_0) \cdot (\mathbf{P}_2 - \mathbf{P}_0) \\ (\mathbf{P}_2 - \mathbf{P}_0) \cdot (\mathbf{P}_1 - \mathbf{P}_0) & (\mathbf{P}_2 - \mathbf{P}_0) \cdot (\mathbf{P}_2 - \mathbf{P}_0) \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} (\mathbf{P}_1 - \mathbf{P}_0) \cdot (\mathbf{X} - \mathbf{P}_0) \\ (\mathbf{P}_2 - \mathbf{P}_0) \cdot (\mathbf{X} - \mathbf{P}_0) \end{bmatrix}$$

Solve for b_1 and b_2 , and then compute $b_0 = 1 - b_1 - b_2$.

A rectangle is most conveniently defined using the equivalent of the center-direction-radius representation for segments. Let $\mathbf{C}$ be the center of the rectangle. Let $\mathbf{D}_0$ and $\mathbf{D}_1$ be perpendicular direction vectors that are parallel to the sides of the rectangle. Let r_0 and r_1 be the *extents* (half-lengths of the sides) of the rectangle. Any point $\mathbf{X}$ on the rectangle is of the form

$$\mathbf{X} = \mathbf{C} + y_0\mathbf{D}_0 + y_1\mathbf{D}_1$$

where $|y_0| \leq r_0$ and $|y_1| \leq r_1$. This representation is least susceptible to numerical round-off errors when testing for parallelism of objects in an intersection query. The plane containing the rectangle has a plane origin $\mathbf{C}$ and a planar normal $\mathbf{N} = \mathbf{D}_0 \times \mathbf{D}_1$.

EXERCISE 10.1

We have seen center-direction-extent representations for segments and rectangles. Formulate a center-direction-extent representation for a triangle. ■

For other planar components, if you have a 2D representation that imposes constraints on the 2-tuples (y_0, y_1) , you automatically have a 3D representation of the object via Equation (10.1), where $y^2 = \mathbf{N} \cdot (\mathbf{X} - \mathbf{P})$.

10.3 BOXES

The simplest box is an *axis-aligned box*, usually called an *axis-aligned bounding box* (AABB) because of its frequent use as a bounding volume. An AABB is defined by two extreme points, $\mathbf{P}_{\min} = (x_{\min}, y_{\min}, z_{\min})$ and $\mathbf{P}_{\max} = (x_{\max}, y_{\max}, z_{\max})$. The box and its interior consists of points (x, y, z) for which $x_{\min} \leq x \leq x_{\max}$, $y_{\min} \leq y \leq y_{\max}$, and $z_{\min} \leq z \leq z_{\max}$.

An *oriented box* or *oriented bounding box* (OBB) is defined by a center $\mathbf{C}$, three orthonormal axes $\mathbf{D}_i$ that form a right-handed set (mutually perpendicular, $\mathbf{D}_2 = \mathbf{D}_0 \times \mathbf{D}_1$), and three extents $r_i > 0$ for $0 \leq i \leq 2$. Any point $\mathbf{X}$ in the box is of the form

$$\mathbf{X} = \mathbf{C} + y_0\mathbf{D}_0 + y_1\mathbf{D}_1 + y_2\mathbf{D}_2$$

where $|y_0| \leq r_0$, $|y_1| \leq r_1$, and $|y_2| \leq r_2$. An AABB is a special case of an OBB when $\mathbf{D}_0 = (1, 0, 0)$, $\mathbf{D}_1 = (0, 1, 0)$, and $\mathbf{D}_2 = (0, 0, 1)$. If $\mathbf{C} = (c_0, c_1, c_2)$, then the extreme points are $\mathbf{P}_{\min} = (c_0 - r_0, c_1 - r_1, c_2 - r_2)$ and $\mathbf{P}_{\max} = (c_0 + r_0, c_1 + r_1, c_2 + r_2)$.

10.4 QUADRICS

The objects discussed here are examples of *quadric surfaces*, all defined implicitly by a quadratic equation $Q(\mathbf{X}) = 0$.

10.4.1 SPHERES

A *sphere* is defined by the set of all points $\mathbf{X}$ equidistant from a center point $\mathbf{C}$ with distance $r > 0$ (the radius). The quadratic equation defining the set is $|\mathbf{X} - \mathbf{C}|^2 = r^2$. If $\mathbf{X} = (x_0, x_1, x_2)$ and $\mathbf{C} = (c_0, c_1, c_2)$, then the quadratic equation is

$$(x_0 - c_0)^2 + (x_1 - c_1)^2 + (x_2 - c_2)^2 = r^2$$

10.4.2 ELLIPSOIDS

An *ellipsoid* in standard axis-aligned form has its center at the origin $(0, 0, 0)$ and its axes aligned with the Cartesian coordinate axes. The half-lengths of the ellipsoid are a_0 in the direction $(1, 0, 0)$, a_1 in the direction $(0, 1, 0)$, and a_2 in the direction $(0, 0, 1)$. If $\mathbf{X} = (x_0, x_1, x_2)$ is a point on the ellipsoid, then

$$\left(\frac{x_0}{a_0}\right)^2 + \left(\frac{x_1}{a_1}\right)^2 + \left(\frac{x_2}{a_2}\right)^2 = 1$$

The representation for an ellipsoid similar to the center-direction-extent for an oriented box is as follows. Given a center point $\mathbf{C}$ and orthonormal axis directions $\mathbf{U}_i$ for $0 \leq i \leq 2$, the ellipsoid is represented by

$$(\mathbf{X} - \mathbf{C})^T \mathbf{R} \mathbf{D} \mathbf{R}^T (\mathbf{X} - \mathbf{C}) = 1$$

where $\mathbf{R} = [\mathbf{U}_0 \ \mathbf{U}_1 \ \mathbf{U}_2]$ is a rotation matrix whose columns are the specified direction vectors, $\mathbf{D} = \text{Diag}\{1/a_0^2, 1/a_1^2, 1/a_2^2\}$ has positive diagonal entries that are the squared

half-lengths, and $\mathbf{X}$ is any point on the ellipsoid. This representation is obtained by thinking of the point representation

$$\mathbf{X} = \mathbf{C} + y_0 \mathbf{U}_0 + y_1 \mathbf{U}_1 + y_2 \mathbf{U}_2$$

where $y_i = \mathbf{U}_i \cdot (\mathbf{X} - \mathbf{C})$. Think of (y_0, y_1, y_2) as a point on an ellipsoid in standard form:

$$\begin{aligned} 1 &= \left(\frac{y_0}{a_0}\right)^2 + \left(\frac{y_1}{a_1}\right)^2 + \left(\frac{y_2}{a_2}\right)^2 \\ &= \left(\frac{\mathbf{U}_0 \cdot (\mathbf{X} - \mathbf{C})}{a_0}\right)^2 + \left(\frac{\mathbf{U}_1 \cdot (\mathbf{X} - \mathbf{C})}{a_1}\right)^2 + \left(\frac{\mathbf{U}_2 \cdot (\mathbf{X} - \mathbf{C})}{a_2}\right)^2 \\ &= \frac{(\mathbf{X} - \mathbf{C})^T \mathbf{U}_0 \mathbf{U}_0^T (\mathbf{X} - \mathbf{C})}{a_0^2} + \frac{(\mathbf{X} - \mathbf{C})^T \mathbf{U}_1 \mathbf{U}_1^T (\mathbf{X} - \mathbf{C})}{a_1^2} + \frac{(\mathbf{X} - \mathbf{C})^T \mathbf{U}_2 \mathbf{U}_2^T (\mathbf{X} - \mathbf{C})}{a_2^2} \\ &= (\mathbf{X} - \mathbf{C})^T \left(\frac{\mathbf{U}_0 \mathbf{U}_0^T}{a_0^2} + \frac{\mathbf{U}_1 \mathbf{U}_1^T}{a_1^2} + \frac{\mathbf{U}_2 \mathbf{U}_2^T}{a_2^2} \right) (\mathbf{X} - \mathbf{C}) \\ &= (\mathbf{X} - \mathbf{C})^T \begin{bmatrix} \mathbf{U}_0 & \mathbf{U}_1 & \mathbf{U}_2 \end{bmatrix} \begin{bmatrix} 1/a_0^2 & 0 & 0 \\ 0 & 1/a_1^2 & 0 \\ 0 & 0 & 1/a_2^2 \end{bmatrix} \begin{bmatrix} \mathbf{U}_0^T \\ \mathbf{U}_1^T \\ \mathbf{U}_2^T \end{bmatrix} (\mathbf{X} - \mathbf{C}) \\ &= (\mathbf{X} - \mathbf{C})^T \mathbf{R} \mathbf{D} \mathbf{R}^T (\mathbf{X} - \mathbf{C}) \end{aligned}$$

The most general form for the ellipsoid is $\mathbf{X}^T \mathbf{A} \mathbf{X} + \mathbf{B}^T \mathbf{X} + c = 0$, where A is a positive definite matrix. The fancy term *positive definite* means that A is a symmetric matrix that can be factored into $A = \mathbf{R} \mathbf{D} \mathbf{R}^T$, where $\mathbf{R}$ is a rotation matrix and $\mathbf{D}$ is a diagonal matrix whose diagonal entries are positive. It is possible to algebraically manipulate the quadratic equation, analogous to completing the square for a quadratic polynomial of one variable, and obtain the equation

$$(\mathbf{X} - \mathbf{C})^T \mathbf{M} (\mathbf{X} - \mathbf{C}) = 1$$

The center is $\mathbf{C} = -\mathbf{A}^{-1} \mathbf{B} / 2$, and the matrix $\mathbf{M}$ is

$$\mathbf{M} = \frac{\mathbf{A}}{\mathbf{B}^T \mathbf{A}^{-1} \mathbf{B} / 4 - c}$$

The matrix $\mathbf{M}$ can itself be factored into $\mathbf{M} = \mathbf{R} \mathbf{D} \mathbf{R}^T$ using an eigendecomposition; see Section 16.2.

10.4.3 CYLINDERS

An *infinite cylinder* is the set of all points a distance r from a line $\mathbf{P} + t\mathbf{D}$, where $\mathbf{D}$ is unit length and where t is any real number. You may construct the cylinder points $\mathbf{X}$ by selecting two vectors $\mathbf{U}$ and $\mathbf{V}$ so that the set $\{\mathbf{D}, \mathbf{U}, \mathbf{V}\}$ is an orthonormal set. The parametric representation is

$$\mathbf{X}(t, \theta) = \mathbf{P} + t\mathbf{D} + r((\cos \theta)\mathbf{U} + (\sin \theta)\mathbf{V})$$

for $\theta \in [0, 2\pi)$. Notice that $|(\cos \theta)\mathbf{U} + (\sin \theta)\mathbf{V}| = 1$ for all θ , so the representation says to start at the line point $\mathbf{P} + t\mathbf{D}$, and then walk radially outward by r units in any direction specified by θ .

To obtain an implicit representation as a quadratic equation, observe that $\mathbf{U} \cdot (\mathbf{X} - \mathbf{P}) = r \cos \theta$ and $\mathbf{V} \cdot (\mathbf{X} - \mathbf{P}) = r \sin \theta$. Thus,

$$\begin{aligned} r^2 &= (r \cos \theta)^2 + (r \sin \theta)^2 \\ &= (\mathbf{U} \cdot (\mathbf{X} - \mathbf{P}))^2 + (\mathbf{V} \cdot (\mathbf{X} - \mathbf{P}))^2 \\ &= (\mathbf{X} - \mathbf{P})^T (\mathbf{U}\mathbf{U}^T + \mathbf{V}\mathbf{V}^T) (\mathbf{X} - \mathbf{P}) \\ &= (\mathbf{X} - \mathbf{P})^T (I - \mathbf{D}\mathbf{D}^T) (\mathbf{X} - \mathbf{P}) \end{aligned}$$

where I is the identity matrix. I have used the fact that for an orthonormal set of vectors, $I = \mathbf{D}\mathbf{D}^T + \mathbf{U}\mathbf{U}^T + \mathbf{V}\mathbf{V}^T$. In summary, the cylinder is an implicit surface defined by

$$(\mathbf{X} - \mathbf{P})^T (I - \mathbf{D}\mathbf{D}^T) (\mathbf{X} - \mathbf{P}) = r^2$$

This equation is independent of how you chose $\mathbf{U}$ and $\mathbf{V}$, which in hindsight should be apparent because of the symmetry of the cylinder.

EXERCISE
10.2

If $\{\mathbf{D}, \mathbf{U}, \mathbf{V}\}$ is an orthonormal set of vectors, prove that $I = \mathbf{D}\mathbf{D}^T + \mathbf{U}\mathbf{U}^T + \mathbf{V}\mathbf{V}^T$. *Hint:* Write a point $\mathbf{X}$ as a combination of the orthonormal vectors and factor the equation. ■

A *finite cylinder* is a truncated infinite cylinder, where $|t| \leq h/2$ for a specified height h . We will refer to finite cylinders simply as “cylinders.” If we need to talk about infinite cylinders, we will refer to them explicitly as “infinite cylinders.”

10.4.4 CONES

A *single-sided cone* is usually defined as follows. Let the cone axis be a ray $\mathbf{P} + t\mathbf{D}$ for $t \geq 0$. The cone vertex is the point $\mathbf{P}$. The cone wall is the set of points at an angle θ

from the cone axis. If $\mathbf{X}$ is on the cone wall, then

$$\mathbf{D} \cdot \left(\frac{\mathbf{X} - \mathbf{P}}{|\mathbf{X} - \mathbf{P}|} \right) = \cos \theta$$

Normally we work with cones whose angle is acute, namely, $\theta \in (0, \pi/2)$. A quadratic equation may be constructed for which the cone is defined implicitly. Multiplying the previously displayed equation by $|\mathbf{X} - \mathbf{P}|$ yields

$$\mathbf{D} \cdot (\mathbf{X} - \mathbf{P}) = (\cos \theta) |\mathbf{X} - \mathbf{P}|$$

Now square the equation and use the fact that for any vector, $|\mathbf{V}|^2 = \mathbf{V} \cdot \mathbf{V}$,

$$(\mathbf{X} - \mathbf{P})^T \mathbf{D} \mathbf{D} (\mathbf{X} - \mathbf{P}) = (\mathbf{D} \cdot (\mathbf{X} - \mathbf{P}))^2 = (\cos \theta)^2 (\mathbf{X} - \mathbf{P}) \cdot (\mathbf{X} - \mathbf{P})$$

Subtract the right-hand-side term from the equation and factor to obtain

$$(\mathbf{X} - \mathbf{P})^T (\mathbf{D} \mathbf{D}^T - (\cos \theta)^2 \mathbf{I}) (\mathbf{X} - \mathbf{P}) = 0$$

where $\mathbf{I}$ is the identity matrix.

The process of squaring introduces additional solutions. In fact, the quadratic equation defines a *double-sided cone*. The portion on the original single-sided cone is obtained by the additional constraint $\mathbf{D} \cdot (\mathbf{X} - \mathbf{P}) \geq 0$ (choose only those points forming an acute angle with $\mathbf{D}$).

Just as an infinite cylinder may be truncated to a finite one, an infinite single-side cone may be truncated by a plane perpendicular to the cone axis. Sometimes this is called a *capped cone*.

EXERCISE 10.3

Prove that any point on the cone is of the parametric form

$$\mathbf{X}(t, \phi) = \mathbf{P} + t\mathbf{D} + (t \tan \theta)(\cos \phi \mathbf{U} + \sin \phi \mathbf{V})$$

where $\{\mathbf{D}, \mathbf{U}, \mathbf{V}\}$ is an orthonormal set. The parameters are constrained to $t \geq 0$ and $\phi \in [0, 2\pi)$. ■

10.5 SPHERE-SWEPT VOLUMES

Sphere-swept volumes have become popular because they sometimes lead to simpler intersection tests than those for other bounding volumes. The idea of a sphere-swept volume is that you construct a volume by placing the center $\mathbf{C}$ of a sphere of a radius r at each and every point of a set called a *medial set*. Any point contained in any of these spheres is part of the volume. The formulation of the volume as a set is actually based on distance. If V is a sphere-swept volume for a sphere of radius r , and if M is the medial set, then $\mathbf{X} \in V$ as long as there is a point $\mathbf{P} \in M$ for which $|\mathbf{X} - \mathbf{P}| \leq r$.

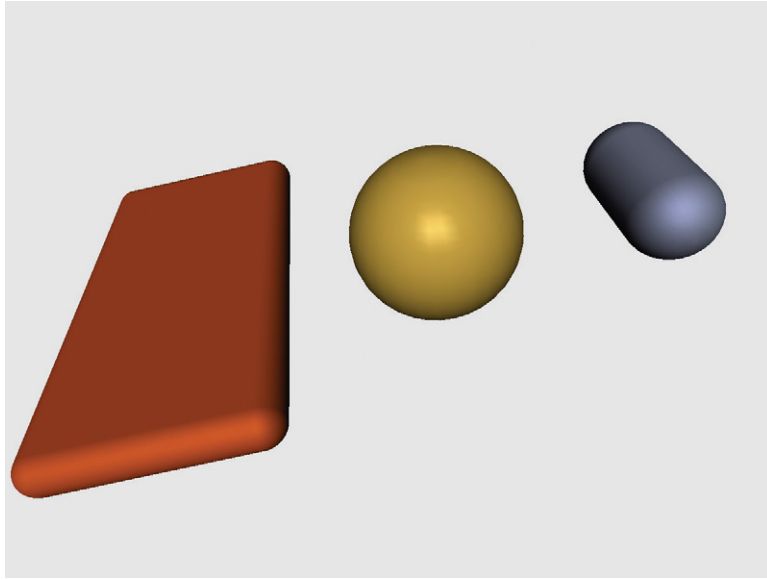


Figure 10.1 Left: A lozenge whose medial set is a rectangle of width 2 and height 4 and whose radius is 0.5. Middle: A sphere of radius 2. Right: A capsule whose medial set is a segment of length 2 and whose radius is 1.

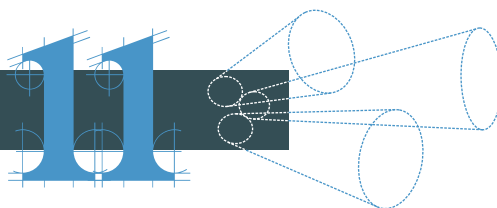
10.5.1 CAPSULES

A *capsule* is a sphere-swept volume where M is a segment. That is, the capsule is the set of all points that are at most a distance r from the line segment.

10.5.2 LOZENGES

A *lozenge* is a sphere-swept volume where M is a rectangle. That is, the lozenge is the set of all points that are at most a distance r from the rectangle. Figure 10.1 shows a sphere, a capsule, and a lozenge. Naturally, you can define other types of sphere-swept volumes. For example, a torus is a sphere-swept volume whose medial set is a circle. An oriented box with its corner regions replaced by octants of a sphere is a sphere-swept volume whose medial set is a smaller oriented box. And a tube surface is the boundary of a sphere-swept volume whose medial set is any curve of your choosing.

CHAPTER



CURVES

At first glance, curves do not appear to be a central topic in building a game engine. Many game engines concentrate on taking polygonal models and processing them for display by the renderer. If objects must change position or orientation during game play, the standard approach has been just to move the objects in a simple fashion, using translation by constant vector offset and rotation by a constant angle—something that requires only vector and matrix algebra (i.e., linear algebra, emphasis on *line*). But curves are actually quite useful when you think about it. For example, if a flight simulator wishes to support realistic flight dynamics, such as the correct banking of a plane as it makes a tight turn, curves can be of assistance. The bank angle is related to how much *bending* there is in the curve that represents the flight path, requiring the concept of *curvature of a curve*. Moreover, if the plane is required to travel at a constant speed along the curved path, the calculations involve knowing something about arc length and the concept of *reparameterization by arc length*.

Another popular example is in the construction of a game that requires tunnels. Many developers are interested in specifying the central curve of a tunnel and the width of the tunnel along that curve. From this information the tunnel walls can be built as a polygon mesh. An understanding of the theory of curves is essential in this construction.

Finally, curved surfaces have become quite popular, if not essential, for building content in a game that is more realistic-looking than the standard polygonal content. The content is typically dynamically tessellated during game play. An understanding of the theory of curves will be quite useful because the same ideas extend to surfaces—the ideas in tessellating curves apply equally well to tessellating surfaces. Understanding curves is a prerequisite to understanding surfaces.

The topic of curves is quite extensive, and only a brief summary is given in this chapter. The basic concepts that are covered in Sections 11.1 and 11.2 are arc length, reparameterization by arc length, curvature, torsion, tangents, normals, and binormals. Special classes of curves are considered in Sections 11.3 through 11.7: Bézier curves; natural, clamped, and closed cubic splines; B-spline curves; nonuniform rational B-spline (NURBS) curves; and tension-continuity-bias splines. Topics discussed here that are less frequently found in the standard references are subdivision of a curve by various methods (by uniform sampling in curve parameter, by arc length, by midpoint distance) and fast recursive subdivision for cubic curves, all considered in Section 11.8. I use the term *parametric subdivision* to distinguish this topic from *subdivision curves* or *subdivision surfaces*. Finally, orientation of moving objects along a curved path is discussed in Section 11.9. This is useful for applications such as flight simulators, where the orientation of the airplanes must be physically realistic.

11.1 DEFINITIONS

A *parametric curve* is a function $\mathbf{X} : [t_{\min}, t_{\max}] \subset \mathbb{R} \rightarrow \mathbb{R}^n$. The curve endpoints are $\mathbf{X}(t_{\min})$ and $\mathbf{X}(t_{\max})$. Tangent vectors to the curve are $\mathbf{X}'(t)$, the derivative with respect to t . The *forward* (*backward*) direction of traversal is that direction implied by increasing (decreasing) t . The *speed* of traversal is $|\mathbf{X}'(t)|$. A curve $\mathbf{X}(s)$ is said to be *parameterized by arc length* s if $\mathbf{T}(s) = \mathbf{X}'(s)$ is unit length. The relationship between s and t is

$$s(t) = \int_{t_{\min}}^t |\mathbf{X}'(\tau)| d\tau$$

The *length* of the curve is $L = s(t_{\max})$.

A *planar curve* $\mathbf{X}(t) = (x_0(t), x_1(t))$ has associated with it an orthonormal *coordinate frame* given by the tangent vector $\mathbf{T}(s)$ and a normal vector $\mathbf{N}(s)$. The frame relationships are

$$\mathbf{T}'(s) = \kappa(s)\mathbf{N}(s)$$

$$\mathbf{N}'(s) = -\kappa(s)\mathbf{T}(s)$$

The quantity $\kappa(s)$ is called *curvature*. A curve is uniquely determined (modulo rigid motions) by specifying a curvature function.

If $\mathbf{T}(s) = (\cos \theta(s), \sin \theta(s))$, then the normal can be chosen as $\mathbf{N}(s) = (-\sin \theta(s), \cos \theta(s))$. In this case, $\kappa = d\theta/ds$. In terms of the t -components of the curve, curvature is

$$\kappa = \frac{x_0'x_1'' - x_0''x_1'}{((x_0')^2 + (x_1')^2)^{3/2}}$$

A space curve $\mathbf{X}(t) = (x_0(t), x_1(t), x_2(t))$ has associated with it an orthonormal coordinate frame called the *Frenet frame*, given by the tangent vector $\mathbf{T}(s)$, a normal vector $\mathbf{N}(s)$, and a binormal vector $\mathbf{B}(s)$. The frame relationships are called the *Frenet-Serret equations*:

$$\mathbf{T}'(s) = \kappa(s)\mathbf{N}(s)$$

$$\mathbf{N}'(s) = -\kappa(s)\mathbf{T}(s) + \tau(s)\mathbf{B}(s)$$

$$\mathbf{B}'(s) = -\tau(s)\mathbf{N}(s)$$

The quantity $\kappa(s)$ is the curvature and the quantity $\tau(s)$ is called *torsion*. A curve is uniquely determined (modulo rigid motions) by specifying both a curvature function and a torsion function.

In terms of the t -components of the curve, curvature is

$$\kappa = \frac{|\mathbf{X}' \times \mathbf{X}''|}{|\mathbf{X}'|^3}$$

The torsion is

$$\tau = \frac{\mathbf{X}' \cdot (\mathbf{X}'' \times \mathbf{X}''')}{|\mathbf{X}' \times \mathbf{X}''|^2}.$$

The curve normal is

$$\mathbf{N} = \frac{(\mathbf{X}' \cdot \mathbf{X}')\mathbf{X}'' - (\mathbf{X}' \cdot \mathbf{X}'')\mathbf{X}'}{|\mathbf{X}'||\mathbf{X}' \times \mathbf{X}''|}$$

Observe that the normal vector is not defined when the denominator is zero. This occurs when the speed is zero, $|\mathbf{X}'| = 0$, or when the velocity and acceleration are parallel, $\mathbf{X}' \times \mathbf{X}'' = \mathbf{0}$.

11.2 REPARAMETERIZATION BY ARC LENGTH

Given a curve $\mathbf{X}(t)$ for $t \in [t_{\min}, t_{\max}]$, it may be desirable to evaluate curve quantities (position, coordinate frame, curvature, torsion) by specifying an arc length $s \in [0, L]$, where L is the total length of the curve. The algorithm requires computing $t \in [t_{\min}, t_{\max}]$ that corresponds to the specified s . This is accomplished by a numerical inversion of the integral equation relating s to t . Define $\text{Speed}(t) = |\mathbf{X}'(t)|$ and $\text{Length}(t) = \int_{t_{\min}}^t |\mathbf{X}'(\tau)| d\tau$. The problem is now to solve $\text{Length}(t) - s = 0$ for the specified s , a root-finding task. From the definition of arc length, the root must be unique. An application of Newton's method will suffice (see Section 16.5.1). Evaluation of $\text{Length}(t)$ does require numerical integration. Romberg integration or

Gaussian quadrature works fine in this setting (see Section 16.6). The pseudocode for the algorithm is

```

Input:  tmin, tmax, L, s in [0,L]
Output: t in [tmin,tmax] corresponding to s

// Choose an initial guess based on relative location of s in [0,L].
ratio = s/L;
t = (1-ratio)*tmin + ratio*tmax;

// Compute Newton iterates to search for the root.
for (i = 0; i < IMAX; i++)
{
    diff = Length(t) - s;
    if (|diff| < EPSILON)
    {
        return t;
    }
    t -= diff/Speed(t);
}

// Newton's method failed to converge. Return your best guess.
return t;

```

An application must choose the maximum number of iterations IMAX and a tolerance EPSILON for how close to zero the root is. These will depend on which functions you decide to use to search for roots.

Two frequently asked questions on the Usenet and game developer news forums are

1. How do I select points on a curve that are equally spaced along the curve?
2. How do I move a camera with constant speed along a curved path?

The answer to both questions is use reparameterization by arc length. The first question essentially asks how to compute the points on a curve given a uniformly spaced set of arc lengths. The second question is really the same one, but in disguise. If you have a set of points equally spaced by arc length, you can move the camera from one point to the next on each frame, leading to a smooth motion with constant speed.

11.3 BÉZIER CURVES

Bézier curves are popular with game programmers for their mathematical simplicity and ease of use.

11.3.1 DEFINITIONS

Given an ordered list of three-dimensional control points $\mathbf{P}_i$ for $0 \leq i \leq n$, the Bézier curve for the points is

$$\mathbf{X}(t) = \sum_{i=0}^n B_{n,i}(t) \mathbf{P}_i$$

for $t \in [0, 1]$ and where the coefficients of the control points are the Bernstein polynomials

$$B_{n,i}(t) = C(n; i) t^i (1 - t)^{n-i} \quad (11.1)$$

with combinatorial values $C(n; i) = n! / (i!(n - i)!)$. The barycentric form of the curve is

$$\mathbf{X}(u, v) = \sum_{i+j=n} C(n; i, j) u^i v^j \mathbf{Q}_{i,j}$$

where $u \in [0, 1]$, $v \in [0, 1]$, $u + v = 1$, $i \geq 0$, $j \geq 0$, and $\mathbf{Q}_{i,j} = \mathbf{P}_i$. The formula appears to be bivariate, but the condition $v = 1 - u$ shows that it is in fact univariate. The derivative of a Bézier curve is

$$\mathbf{X}'(t) = n \sum_{i=0}^{n-1} B_{n-1,i}(t) (\mathbf{P}_{i+1} - \mathbf{P}_i)$$

11.3.2 EVALUATION

In evaluating a Bézier curve, a decision must be made about whether speed or accuracy is more important. For real-time applications, speed is usually the important criterion. Inaccuracies in the computed positions are not noticeable in the sampled curve.

Using the Bernstein form of a Bézier curve, the Bernstein polynomials are evaluated first. The polynomials are computed for the selected t and for all values $0 \leq i \leq n$. The control points are then multiplied by the coefficients and summed. Assuming a fixed degree n and assuming that the combinatorial values $C(n; i)$ are precomputed, the number of multiplications required to compute each polynomial coefficient is

n . For small degree n , the number of multiplications can be reduced by computing intermediate products of powers of t and $1 - t$, but this optimization is not considered at the moment in the operation count. Multiplying a polynomial coefficient times control point requires three multiplications. Given $n + 1$ terms, the number of required multiplications is $(n + 1)(n + 3)$. There are $n + 1$ 3D terms to sum for a total of $3n$ additions. The total operation count for a single Bézier curve evaluation is $n^2 + 7n + 3$ operations.

Using the barycentric form of a Bézier curve, evaluation is possible by using the de Casteljau algorithm (a good reference on the topic is [Far90]). Define $\mathbf{Q}_{i,j}^0 = \mathbf{Q}_{i,j}$ to be the original control points. The algorithm is

$$\mathbf{Q}_{i,j}^r(u, v) = u\mathbf{Q}_{i+1,j}^{r-1} + v\mathbf{Q}_{i,j+1}^{r-1}$$

for $1 \leq r \leq n$ and $i + j = n - r$. For each r there are six multiplications and three additions on the right-hand side of the equation. The number of terms to compute for each r is $n - r$. Total operation count for a single evaluation is $9n(n - 1)/2$ operations. This is quadratic order, just as for the Bernstein evaluation, but the constant is 9 rather than 1. However, the de Casteljau algorithm is numerically stable, whereas the Bernstein form is not, particularly for large n . The amount of numerical error in the Bernstein form is visually insignificant for rendering purposes for small degree $n \leq 4$. The savings in time is clearly worth using the Bernstein form.

11.3.3 DEGREE ELEVATION

A Bézier curve with $n + 1$ control points is a polynomial of degree n . An equivalent Bézier curve with $n + 2$ control points, and that is a polynomial of degree $n + 1$, can be constructed. The process, called *degree elevation*, is useful in smoothly piecing together Bézier curves. The degree-elevated Bézier curve is obtained by multiplying the Bernstein form of the curve by $1 = (t + (1 - t))$. The multiplication by 1 does not intrinsically change the curve, but the polynomial coefficients are changed because of the multiplication by $(t + (1 - t))$. The degree-elevated curve is

$$\mathbf{x}(t) = \sum_{i=0}^{n+1} B_{n+1,i}(t) \left[\left(1 - \frac{i}{n+1}\right) \mathbf{p}_i + \frac{i}{n+1} \mathbf{p}_{i-1} \right]$$

11.3.4 DEGREE REDUCTION

If the original Bézier curve is quadratic, the degree-elevated curve is cubic, showing that there are some cubic curves that can be represented by quadratic curves. However, not all cubic curves are representable by quadratic curves. For example, a cubic curve that is S-shaped cannot be represented by a quadratic curve. It may be desirable

to reduce the degree on a Bézier curve so that the curve evaluations are less expensive to compute. Although it is not always possible to get an exact degree-reduced representation, it is possible to build one that approximately fits the curve. A least-squares fit can be used to obtain the degree-reduced curve. If the original curve has a lot of variation, the least-squares fit may not be as good a fit as is desired. For example, if the original curve is cubic and has control points $(-2, 0, 0)$, $(-1, 1, 0)$, $(1, -1, 0)$, and $(2, 0, 0)$, the curve is S-shaped. The least-squares fit will produce a quadratic curve with control points $(-2, 0, 0)$, $(0, 0, 0)$, and $(2, 0, 0)$. This curve is a straight line segment.

Let the original curve be

$$\mathbf{X}(t) = \sum_{i=0}^n B_{n,i}(t) \mathbf{P}_i$$

and let the degree-reduced curve be

$$\mathbf{Y}(t) = \sum_{i=0}^m B_{m,i}(t) \mathbf{Q}_i$$

where $m < n$. The end control points are required to be the same, $\mathbf{Q}_0 = \mathbf{P}_0$ and $\mathbf{Q}_m = \mathbf{P}_n$. The remaining control points $\mathbf{Q}_i$, $1 \leq i \leq m-1$, are chosen to minimize the integral of the squared differences of the two curves,

$$E(\mathbf{Q}_1, \dots, \mathbf{Q}_{m-1}) = \int_0^1 |\mathbf{X}(t) - \mathbf{Y}(t)|^2 dt$$

The values of the interior control points are determined by setting all the partial derivatives of E to zero, $\partial E / \partial \mathbf{Q}_j = 0$ for $1 \leq j \leq m-1$. This leads to the $m-1$ equations in the $m-1$ unknown control points:

$$\sum_{i=0}^m \frac{(2m+1)C(m; i)}{C(2m; i+j)} \mathbf{Q}_i = \sum_{i=0}^n \frac{(m+n+1)C(n; i)}{C(m+n; i+j)} \mathbf{P}_i$$

The system always has a solution.

The equations can be solved symbolically for some cases of interest. For $n = 3$ and $m = 2$, the solution is

$$\mathbf{Q}_0 = \mathbf{P}_0$$

$$\mathbf{Q}_1 = \frac{1}{4} (-\mathbf{P}_0 + 3\mathbf{P}_1 + 3\mathbf{P}_2 - \mathbf{P}_3)$$

$$\mathbf{Q}_2 = \mathbf{P}_3$$

For $n = 4$ and $m = 3$, the solution is

$$\mathbf{Q}_0 = \mathbf{P}_0$$

$$\mathbf{Q}_1 = \frac{1}{42} (-11\mathbf{P}_0 + 44\mathbf{P}_1 + 18\mathbf{P}_2 - 12\mathbf{P}_3 + 3\mathbf{P}_4)$$

$$\mathbf{Q}_2 = \frac{1}{42} (3\mathbf{P}_0 - 12\mathbf{P}_1 + 18\mathbf{P}_2 + 44\mathbf{P}_3 - 11\mathbf{P}_4)$$

$$\mathbf{Q}_3 = \mathbf{P}_4$$

For $n = 4$ and $m = 2$, the solution is

$$\mathbf{Q}_0 = \mathbf{P}_0$$

$$\mathbf{Q}_1 = \frac{1}{28} (-11\mathbf{P}_0 + 16\mathbf{P}_1 + 18\mathbf{P}_2 + 16\mathbf{P}_3 - 11\mathbf{P}_4)$$

$$\mathbf{Q}_2 = \mathbf{P}_4$$

11.4 NATURAL, CLAMPED, AND CLOSED CUBIC SPLINES

These curve types have the property of *exact interpolation*—the curves pass through all of the sample points. The motivation is based on interpolation of a univariate function. A good discussion of the topic for natural and clamped splines is [BF01]. The closed-spline algorithm is not mentioned in [BF01], but can be developed in a similar manner as the natural and clamped versions. A brief discussion is given here.

A list of points (t_i, f_i) for $0 \leq i \leq n$ is specified. On each interval $[t_i, t_{i+1}]$ with $0 \leq i \leq n-1$, a cubic function $S_i(t) = a_i + b_i(t - t_i) + c_i(t - t_i)^2 + d_i(t - t_i)^3$ is required so that the following conditions are met. The first set of conditions are for exact interpolation:

$$S_i(t_i) = f_i, \quad 0 \leq i \leq n-1, \quad S_{n-1}(t_n) = f_n \quad (11.2)$$

for a total of $n+1$ constraints. The second set requires that the polynomial values at the interior control points must match:

$$S_{i+1}(t_{i+1}) = S_i(t_{i+1}), \quad 0 \leq i \leq n-2 \quad (11.3)$$

for a total of $n-1$ constraints. The third set of conditions requires that the first derivatives at the interior control points must match:

$$S'_{i+1}(t_{i+1}) = S'_i(t_{i+1}), \quad 0 \leq i \leq n-2 \quad (11.4)$$

for a total of $n - 1$ constraints. The fourth set of conditions requires that the second derivatives at the interior control points must match:

$$S''_{i+1}(t_{i+1}) = S''_i(t_{i+1}), \quad 0 \leq i \leq n - 2 \quad (11.5)$$

for a total of $n - 1$ constraints. All conditions together yield $4n - 2$ constraints. The unknown quantities are the coefficients a_i , b_i , c_i , and d_i for $0 \leq i \leq n - 1$. The number of unknowns is $4n$. Two additional constraints must be posed in hopes of obtaining a linear system of $4n$ equations in $4n$ unknowns. The three cases considered here are the following:

- *Natural splines:* $S''_0(t_0) = 0$ and $S''_{n-1}(t_n) = 0$.
- *Clamped splines:* $S'_0(t_0)$ and $S'_{n-1}(t_n)$ are specified by the application.
- *Closed splines:* $S_0(t_0) = S_{n-1}(t_n)$, $S'_0(t_0) = S'_{n-1}(t_n)$, and $S''_0(t_0) = S''_{n-1}(t_n)$, in which case it is necessary that $f_0 = f_n$. Although these appear to be a set of three additional constraints, not two, the requirement that the input data satisfy $f_0 = f_n$ automatically guarantees that $S_0(t_0) = S_{n-1}(t_n)$ whenever the original exact interpolation constraints are satisfied.

Define $h_i = t_{i+1} - t_i$ for $0 \leq i \leq n - 1$. Equation (11.2) implies

$$\begin{aligned} a_i &= f_i, \quad 0 \leq i \leq n - 1, \\ a_{n-1} + b_{n-1}h_{n-1} + c_{n-1}h_{n-1}^2 + d_{n-1}h_{n-1}^3 &= f_n \end{aligned} \quad (11.6)$$

Equation (11.3) implies

$$a_{i+1} = a_i + b_i h_i + c_i h_i^2 + d_i h_i^3, \quad 0 \leq i \leq n - 2 \quad (11.7)$$

Equation (11.4) implies

$$b_{i+1} = b_i + 2c_i h_i + 3d_i h_i^2, \quad 0 \leq i \leq n - 2, \quad (11.8)$$

And Equation (11.5) implies

$$c_{i+1} = c_i + 3d_i h_i, \quad 0 \leq i \leq n - 2 \quad (11.9)$$

Equation (11.9) can be solved for d_i :

$$d_i = \frac{c_{i+1} - c_i}{3h_i}, \quad 0 \leq i \leq n - 1 \quad (11.10)$$

Replacing Equation (11.10) in Equation (11.7) and solving for b_i yields

$$b_i = \frac{a_{i+1} - a_i}{h_i} - \frac{(2c_i + c_{i+1})h_i}{3}, \quad 0 \leq i \leq n - 2 \quad (11.11)$$

Replacing Equation (11.11) in Equation (11.8) yields

$$h_{i-1}c_{i-1} + 2(h_i + h_{i-1})c_i + h_ic_{i+1} = \frac{3(a_{i+1} - a_i)}{h_i} - \frac{3(a_i - a_{i-1})}{h_{i-1}}, \quad 1 \leq i \leq n-1 \quad (11.12)$$

11.4.1 NATURAL SPLINES

Define $c_n = S''_{n-1}(t_n)/2$. The boundary condition $S''_0(t_0) = 0$ yields

$$c_0 = 0 \quad (11.13)$$

The other condition $S''_{n-1}(t_n) = 0$ yields

$$c_n = 0 \quad (11.14)$$

Equations (11.12), (11.13), and (11.14) form a tridiagonal system of linear equations that can be solved by standard methods in $O(n)$ time.

11.4.2 CLAMPED SPLINES

Let the boundary conditions be $S'_0(t_0) = f'_0$ and $S'_{n-1}(t_n) = f'_n$, where f'_0 and f'_n are specified by the application. These lead to two equations,

$$2h_0c_0 + h_0c_1 = \frac{3(a_1 - a_0)}{h_0} - 3f'_0 \quad (11.15)$$

and

$$h_{n-1}c_{n-1} + 2h_{n-1}c_n = 3f'_n - \frac{3(a_n - a_{n-1})}{h_{n-1}} \quad (11.16)$$

where we define $a_n = f_n = S_{n-1}(t_n)$. Equations (11.12), (11.15), and (11.16) form a tridiagonal system of linear equations that can be solved by standard methods in $O(n)$ time.

11.4.3 CLOSED SPLINES

It is necessary that $f_n = f_0$ to obtain a well-posed system of equations defining the polynomial coefficients. In this case $a_0 = a_n$, where we define $a_n = f_n = S_{n-1}(t_n)$. The boundary condition $S''_0(t_0) = S''_{n-1}(t_n)$ and defined value $c_n = S''_{n-1}(t_n)/2$ imply a constraint,

$$c_0 = c_n \quad (11.17)$$

The boundary condition $S'_0(t_0) = S'_{n-1}(t_n)$ implies

$$b_0 = b_{n-1} + 2c_{n-1}h_{n-1} + 3d_{n-1}h_{n-1}^2$$

We also know that

$$\begin{aligned} b_0 &= \frac{a_1 - a_0}{h_0} - \frac{(2c_0 + c_1)h_0}{3} \\ b_{n-1} &= \frac{a_n - a_{n-1}}{h_{n-1}} - \frac{(2c_{n-1} + c_n)h_{n-1}}{3} \\ d_{n-1} &= \frac{c_n - c_{n-1}}{3h_{n-1}} \end{aligned}$$

Substituting these quantities in the last constraint yields

$$h_{n-1}c_{n-1} + 2(h_{n-1} + h_0)c_0 + h_0c_1 = 3 \left(\frac{a_1 - a_0}{h_0} - \frac{a_0 - a_{n-1}}{h_{n-1}} \right) \quad (11.18)$$

Equations (11.12), (11.17), and (11.18) form a linear system of equations. It is not tridiagonal, but the cyclical nature of the matrix allows you to solve the system in $O(n)$ time.

The natural-, clamped-, and closed-spline interpolations were defined for fitting a sequence of scalar values, but they can be simply extended to curves by fitting each coordinate component of the curve separately.

11.5 B-SPLINE CURVES

The splines of the previous section are exact interpolating and require solving systems of equations whose size is the number of control points. If one of the control points is changed, the system of equations must be solved again and the entire curve is affected by the change. This might be an expensive operation in an interactive application or when the number of control points is very large. An alternative is to obtain *local control* in exchange for a nonexact interpolation. In this setting, changing a control point affects the curve only locally and any recalculations for the curve are minimal.

The control points for a B-spline curve are $\mathbf{B}_i$, $0 \leq i \leq n$. The construction is dimensionless, so the control points can be in whatever dimension interests you. The degree d of the curve must be selected so that $1 \leq d \leq n$. The curve itself is defined by

$$\mathbf{X}(u) = \sum_{i=0}^n N_{i,d}(u) \mathbf{B}_i \quad (11.19)$$

where the functions $N_{i,d}(u)$ are called the *B-spline basis functions*. These functions are defined recursively and require selection of a sequence of scalars u_i for $0 \leq i \leq n + d + 1$. The sequence must be nondecreasing, that is, $u_i \leq u_{i+1}$. Each u_i is referred to as a *knot*, the total sequence a *knot vector*. The basis function that starts the recursive definition is

$$N_{i,0}(u) = \begin{cases} 1, & u_i \leq u < u_{i+1} \\ 0, & \text{otherwise} \end{cases} \quad (11.20)$$

for $0 \leq i \leq n + d$. The recursion itself is

$$N_{i,j}(u) = \frac{u - u_i}{u_{i+j} - u_i} N_{i,j-1}(u) + \frac{u_{i+j+1} - u}{u_{i+j+1} - u_{i+1}} N_{i+1,j-1}(u) \quad (11.21)$$

for $1 \leq j \leq d$ and $0 \leq i \leq n + d - j$. The *support* of a function is the smallest closed interval on which the function has at least one nonzero value. The support of $N_{i,0}(u)$ is clearly $[u_i, u_{i+1}]$. In general, the support of $N_{i,j}(u)$ is $[u_i, u_{i+j+1}]$. We will use this information later to show how $\mathbf{X}(u)$ for a specific value of u depends only on a small number of control points, the indices of those points related to the choice of u . This property is called *local control* and will be important when you want to deform a portion of a curve (or surface) by varying only those control points affecting that portion.

The knots can be within any domain, but I will choose them to be in $[0, 1]$ to provide a standardized interface for B-spline and NURBS curves and surfaces.

11.5.1 TYPES OF KNOT VECTORS

The main classification of the knot vector is that it is either *open* or *periodic*. If open, the knots are either *uniform* or *nonuniform*. Periodic knot vectors have uniformly spaced knots. The use of the term *open* is perhaps a misnomer since you can construct a closed B-spline curve from an open knot vector. The standard way to construct a closed curve uses periodic knot vectors. An open, uniform knot vector is defined by

$$u_i = \begin{cases} 0, & 0 \leq i \leq d \\ \frac{i-d}{n+1-d}, & d+1 \leq i \leq n \\ 1, & n+1 \leq i \leq n+d+1 \end{cases}$$

An open, nonuniform knot vector is in the same format except that the values u_i for $d+1 \leq i \leq n$ are user defined. These must be selected to maintain monotonicity $0 \leq u_{d+1} \leq \dots \leq u_{n+1} \leq 1$. A periodic knot vector is defined by

$$u_i = \frac{i-d}{n+1-d}, \quad 0 \leq i \leq n+d+1$$

Some of the knots are outside the domain $[0, 1]$, but this occurs to force the curve to have period 1. When evaluating $\mathbf{X}(u)$, any input value of u outside $[0, 1]$ is reduced to this interval by periodicity before evaluation of the curve point.

11.5.2 EVALUATION

The straightforward method for evaluation of $\mathbf{X}(u)$ is to compute all of $N_{i,d}(u)$ for $0 \leq i \leq n$ using the recursive formulas from Equations (11.20) and (11.21). The pseudocode to compute the basis function values is shown next. The value n , degree d , knots $u[k]$, and control points $B[k]$ are assumed to be globally accessible.

```
float N (int i, int j, float u)
{
    if (j > 0)
    {
        c0 = (u - u[i]) / (u[i + j] - u[i]);
        c1 = (u[i + j + 1] - u) / (u[i + j + 1] - u[i + 1]);
        return c0 * N(i, j - 1, u) + c1 * N(i + 1, j - 1, u);
    }
    else // j == 0
    {
        return (u[i] <= u && u < u[i + 1] ? 1 : 0);
    }
}

Point X (float u)
{
    Point result = ZERO;
    for (i = 0; i <= n; i++)
    {
        result += N(i, d, u) * B[i];
    }
    return result;
}
```

This is an inefficient algorithm because many of the basis functions are evaluated twice. For example, the value $N_{0,d}(u)$ requires computing $N_{0,d-1}(u)$ and $N_{1,d-1}(u)$. The value $N_{1,d}(u)$ also requires computing $N_{1,d-1}(u)$ as well as $N_{2,d-1}(u)$. The recursive dependencies are illustrated in Table 11.1 for $n = 4$ and $d = 2$. The various types of knot vectors are shown below the table of basis function values.

The rows of knot vectors open with brackets and close with parentheses. These indicate that an evaluation for a specified $u \in [0, 1)$ requires searching for the bounding interval $[u_i, u_{i+1})$ containing u . Only those knots in the bracketed portion need to be

Table 11.1 Recursive dependencies for B-spline basis functions for $n = 4$ and $d = 2$.

	$N_{0,2}$	$N_{1,2}$	$N_{2,2}$	$N_{3,2}$	$N_{4,2}$			
	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘			
	$N_{0,1}$	$N_{1,1}$	$N_{2,1}$	$N_{3,1}$	$N_{4,1}$	$N_{5,1}$		
	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘	↓ ↘		
	$N_{0,0}$	$N_{1,0}$	$N_{2,0}$	$N_{3,0}$	$N_{4,0}$	$N_{5,0}$	$N_{6,0}$	
Open Uniform	0	0	[0	$\frac{1}{3}$	$\frac{2}{3}$	1)	1	1
Open Nonuniform	0	0	[0	u_3	u_4	1)	1	1
Periodic	$-\frac{2}{3}$	$-\frac{1}{3}$	[0	$\frac{1}{3}$	$\frac{2}{3}$	1)	$\frac{4}{3}$	$\frac{5}{3}$

searched. The search returns the index of the left endpoint i , where $d \leq i \leq n$. For an open knot vector, the knots corresponding to other indices are included for padding. For a periodic knot vector, the knots corresponding to other indices are included to force the periodicity.

To avoid the redundant calculations, you might think to evaluate the table from the bottom up rather than from the top down. In our example you would compute $N_{i,0}(u)$ for $0 \leq i \leq 6$ and save these for later access. You would then compute $N_{i,1}(u)$ for $0 \leq i \leq 5$ and look up the values $N_{j,0}(u)$ as needed. Finally, you would compute $N_{i,2}(u)$ for $0 \leq i \leq 4$. The pseudocode follows.

```

Point X (float u)
{
    float basis[d + 1][n + d + 1]; // basis[j][i] = N(i,j)

    for (i = 0; i <= n + d; i++)
    {
        if (u[i] <= u && u < u[i + 1])
        {
            basis[0][i] = 1;
        }
        else
        {
            basis[0][i] = 0;
        }
    }

    for (j = 1; j <= d; j++)
    {
        for (i = 0; i <= n + d - j; i++)

```

Nonzero (boxed) values from Table 11.1 for $N_{3,0}(u) = 1$.

$$\begin{array}{ccccccccccc}
N_{0,2} & & \boxed{N_{1,2}} & & \boxed{N_{2,2}} & & \boxed{N_{3,2}} & & N_{4,2} & & \\
\downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \\
N_{0,1} & & N_{1,1} & & \boxed{N_{2,1}} & & \boxed{N_{3,1}} & & N_{4,1} & & N_{5,1} \\
\downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \\
N_{0,0} & & N_{1,0} & & N_{2,0} & & \boxed{N_{3,0}} & & N_{4,0} & & N_{5,0} & & N_{6,0}
\end{array}$$

```

    {
        c0 = (u - u[i]) / (u[i + j] - u[i]);
        c1 = (u[i + j + 1] - u) / (u[i + j + 1] - u[i + 1]);
        basis[i][j] = c0 * basis[j - 1][i] + c1 * basis[j - 1][i + 1];
    }
}

Point result = ZERO;
for (i = 0; i <= n; i++)
{
    result += basis[d][i] * B[i];
}

return result;
}

```

This is a reasonable modification but still not as efficient as it could be. For a single value of u , only *one* of $N_{i,0}(u)$ is 1; the others are all 0. In our example, suppose that $u \in [u_3, u_4)$ so that $N_{3,0}(u)$ is 1 and all other $N_{i,0}(u)$ are 0. The only nonzero entries from Table 11.1 are shown as boxed quantities in Table 11.2.

The boxed entries cover a triangular portion of the table. The values on the left diagonal edge and on the right vertical edge are computed first since each value effectively depends only on one previous value, the other value already known to be 0. If $N_{i,0}(u) = 1$, the left diagonal edge is generated by

$$N_{i-j,j}(u) = \frac{u_{i+1} - u}{u_{i+1} - u_{i-j+1}} N_{i-j+1,j-1}(u)$$

and the right vertical edge is generated by

$$N_{i,j}(u) = \frac{u - u_i}{u_{i+j} - u_i} N_{i,j-1}(u)$$

both evaluated for $1 \leq j \leq d$. The interior values are computed using the recursive formula, Equation (11.21). The pseudocode for computing the curve point follows.

```

Point X (float u)
{
    float basis[d + 1][n + d + 1]; // basis[j][i] = N(i,j)

    // Get i for which u[i] <= u < u[i + 1].
    i = GetKey(u);

    // Evaluate left diagonal and right vertical edges.
    for (j = 1; j <= d; j++)
    {
        c0 = (u - u[i]) / (u[i + j] - u[i]);
        c1 = (u[i + 1] - u) / (u[i + 1] - u[i - j + 1]);
        basis[j][i] = c0 * basis[j-1][i];
        basis[j][i - j] = c1 * basis[j - 1][i - j + 1];
    }

    // Evaluate interior.
    for (j = 2; j <= d; j++)
    {
        for (k = i - j + 1; k < i; k++)
        {
            c0 = (u - u[k]) / (u[k + j] - u[k]);
            c1 = (u[k + j + 1] - u) / (u[k + j + 1] - u[k + 1]);
            basis[j][k] = c0 * basis[j - 1][k] * fInvD0 +
                c1 * basis[j - 1][k + 1];
        }
    }

    Point result = ZERO;
    for (j = i - d; j <= i; j++)
    {
        result += basis[d][j] * B[j];
    }
    return result;
}

```

The only remaining issue is how to compute index i from the input parameter u . For optimal efficiency, the computation should take into account whether the knot vector is open or periodic, and if open, whether the knots are uniformly or nonuniformly spaced. The pseudocode follows. Observe that the choice is made to clamp u to $[0, 1]$ when the spline is open and to wrap u to $[0, 1]$ when the spline is periodic.

```

int GetKey (float& u) const
{
    if (knot vector is open) // Open splines clamp to [0,1].

```

```

{
    if (u <= 0) { u = 0; return d; }
    if (u >= 1) { u = 1; return n; }
}
else // Periodic splines wrap to [0,1].
{
    if (u < 0 || u > 1) u -= floor(u);
}

int i;
if (knots are uniformly spaced)
{
    i = d + floor((n + 1 - d) * u);
}
else // Knots are nonuniformly spaced.
{
    for (i = d + 1; i <= n + 1; i++) { if ( u < u[i] ) break; }
    i--;
}
return i;
}

```

In all cases the search for the bounding interval $[u_i, u_{i+1}]$ of u produces an index i , for which $d \leq i \leq n$, according to the discussion immediately following Table 11.1.

The basis function data and operations can be encapsulated into a class `BasisFunction` so that a B-spline curve class has a basis function object for the parameter u . For the purpose of curve evaluation, only two public interface functions must exist for a `BasisFunction` class. One function computes the basis function values at u and returns the index i of the nonzero basis value $N_{i,0}(u)$; call it `int Compute(float u)`. The function returns the index i . The `GetKey` function described earlier becomes a nonpublic helper function for `Compute`. Another function is an accessor to the values $N_{i,d}(u)$; call it `float Basis(int i)`. The `BasisFunction` class stores the degree d internally, so only i needs to be passed. The curve evaluator does not need access to basis function values $N_{i,j}(u)$ for $j < d$. The B-spline curve itself can be encapsulated in a class `BSpineCurve`. This class manages the control points $B[]$, knows the degree d of the curve, and has a `BasisFunction` member called `Nu`. The curve evaluator becomes a member function of `BSpineCurve` and is

```

Point BSpineCurve::X (float u)
{
    int i = Nu.Compute(u);
    Point result = ZERO;
    for (int j = i - d; j <= i; j++)

```

```

    {
        result += Nu.Basis(j) * B[j];
    }
    return result;
}

```

11.5.3 LOCAL CONTROL

Our goal is to dynamically modify the control points of the B-spline curve in order to deform only a portion of that curve. If we were to change exactly one control point $\mathbf{B}_j$ in Equation (11.19), what part of the curve is affected? The modified $\mathbf{B}_j$ is blended into the curve equation via the basis function $N_{j,d}(u)$. The curve associated with those parameters u for which this function is not zero is affected by the change. The set of such u is exactly what we called the support of the function, in this case the interval $[u_j, u_{j+d+1}]$. The property such that changing a control point affects only a small portion of the curve is referred to as *local control*.

The practical application of local control is that in drawing the curve, you create a polyline approximation by selecting samples $\bar{u}_k \in [0, 1]$ for $0 \leq k < m$, with $\bar{u}_k < \bar{u}_{k+1}$ for all k . The curve points are $\mathbf{P}_k = \mathbf{X}(\bar{u}_k)$. The polyline consists of the line segments $\langle \mathbf{P}_k, \mathbf{P}_{k+1} \rangle$ for $0 \leq k < m - 1$. If we were to change control point $\mathbf{B}_j$, only some of the line segments would need to be recomputed. Specifically, define $k_{\min}$ and $k_{\max}$ to be the extreme indices for which $\bar{u}_k \in [u_j, u_{j+d+1}]$. The polyline points $\mathbf{P}_k$ for $k_{\min} \leq k \leq k_{\max}$ are the only ones to be recomputed.

11.5.4 CLOSED CURVES

In order to obtain closed curves, additional control points must be included by the curve designer or automatically generated by the B-spline curve implementation. If the latter, and the implementation allows the user to dynamically modify control points, the additional control points must be modified accordingly.

Closing a B-spline curve with an open knot vector is simple. If the curve has control points $\mathbf{B}_i$ for $0 \leq i \leq n$, the first control point must be duplicated, $\mathbf{B}_{n+1} = \mathbf{B}_0$. An additional knot must also be added. The extra knot is automatically calculated for uniformly spaced knots, but the curve designer must specify the extra knot for nonuniformly spaced knots.

Closing a B-spline curve with a periodic knot vector requires the first d control points to be duplicated, $\mathbf{B}_{n+i} = \mathbf{B}_i$ for $0 \leq i < d$. Since a periodic knot vector has uniformly spaced knots, the d additional knots are automatically calculated.

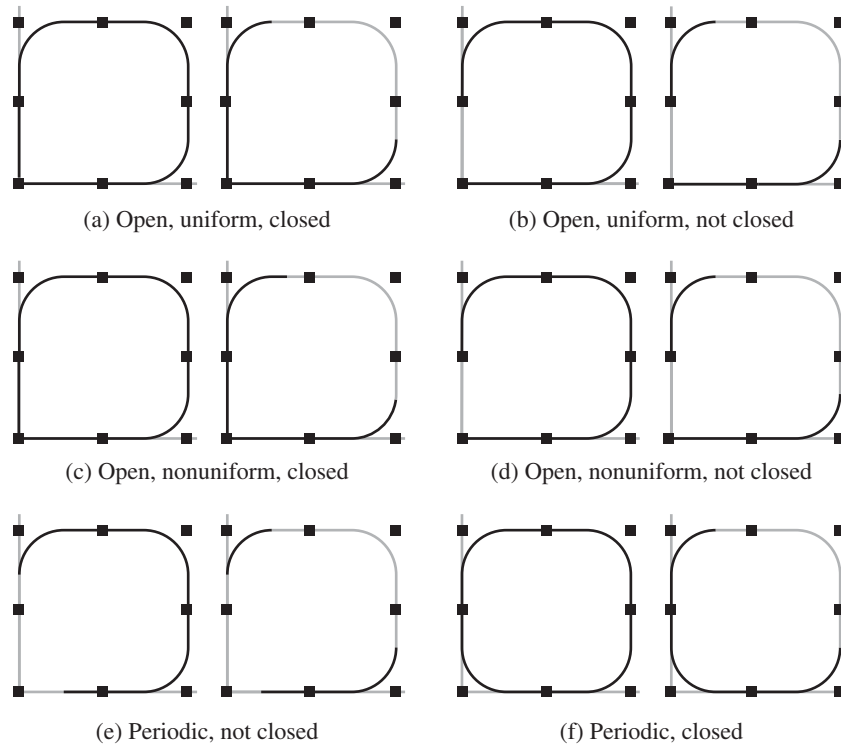


Figure 11.1

Six pairs of B-spline curves of various types. The right image of each pair shows the deformed curve by modifying one control point.

EXAMPLE 11.1

Figure 11.1 shows six pairs of B-spline curves, pairs (a) through (f). The left image in each pair is generated from the eight ordered control points $(0, 0)$, $(1, 0)$, $(2, 0)$, $(2, 1)$, $(2, 2)$, $(1, 2)$, $(0, 2)$, and $(0, 1)$. The right image uses the same control points except that $(2, 2)$ is replaced by $(2.75, 2.75)$. Also, the light gray portions of the curves in the right images are those points that were affected by modifying the control point $(2, 2)$ to $(2.75, 2.75)$. In order to avoid confusion between the two uses of the term *open*, a curve is labeled as either closed or not closed.

Table 11.3 shows the knot vectors and the parameter intervals affected by modifying the control point $(2, 2)$. The nonuniform knot vectors were just chosen arbitrarily. The other knot vectors were automatically generated. ■

Table 11.3 Knot vectors and parameter intervals affected by modifying the control point.

Open, Uniform, Not Closed	$\{0, 0, 0, \frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}, \frac{5}{6}, 1, 1, 1\}$	$[\frac{2}{6}, \frac{5}{6}]$
Open, Nonuniform, Not Closed	$\{0, 0, 0, 0.1, 0.2, 0.4, 0.7, 0.8, 1, 1, 1\}$	$[0.2, 0.8]$
Periodic, Not Closed	$\{-\frac{2}{6}, -\frac{1}{6}, 0, \frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}, \frac{5}{6}, 1, \frac{7}{6}, \frac{8}{6}\}$	$[\frac{2}{6}, \frac{5}{6}]$
Open, Uniform, Closed	$\{0, 0, 0, \frac{1}{7}, \frac{2}{7}, \frac{3}{7}, \frac{4}{7}, \frac{5}{7}, \frac{6}{7}, 1, 1, 1\}$	$[\frac{2}{7}, \frac{5}{7}]$
Open, Nonuniform, Closed	$\{0, 0, 0, 0.1, 0.2, 0.4, 0.7, 0.8, 0.9, 1, 1, 1\}$	$[0.2, 0.8]$
Periodic, Closed	$\{-\frac{2}{8}, -\frac{1}{8}, 0, \frac{1}{8}, \frac{2}{8}, \frac{3}{8}, \frac{4}{8}, \frac{5}{8}, \frac{6}{8}, \frac{7}{8}, 1, \frac{9}{8}, \frac{10}{8}\}$	$[\frac{2}{8}, \frac{5}{8}]$

11.6 NURBS CURVES

NURBS is an acronym for *nonuniform rational B-spline*. B-spline curves are piecewise polynomial functions. The concept of NURBS provides a level of generality by allowing the curves to be piecewise rational polynomial functions; that is, the curve components are ratios of polynomial functions. The mathematics of NURBS is quite deep and is described concisely in [Far99]. Not to deemphasize the theoretical foundations, but for our purposes the use of NURBS is for the greater flexibility in constructing shapes than that provided by B-splines.

The control points for a NURBS curve are $\mathbf{B}_i$ for $0 \leq i \leq n$, just as in the case of B-spline curves. However, *control weights* are also provided, w_i for $0 \leq i \leq n$. The construction is dimensionless; the control points can be m -tuples. The idea for defining NURBS is quite simple. The $(m + 1)$ -tuples $(w_i \mathbf{B}_i, w_i)$ are used to create a B-spline curve $(\mathbf{Y}(u), w(u))$. These tuples are treated as homogeneous coordinates. To project back to m -dimensional space, you divide by the last component: $\mathbf{X}(u) = \mathbf{Y}(u)/w(u)$. The degree d of the curve is selected so that $1 \leq d \leq n$. The NURBS curve is defined by

$$\mathbf{X}(u) = \frac{\sum_{i=0}^n N_{i,d}(u) w_i \mathbf{B}_i}{\sum_{i=0}^n N_{i,d}(u) w_i} \quad (11.22)$$

where $N_{i,d}(u)$ are the B-spline basis functions discussed earlier.

EXAMPLE 11.2

The classical example of the greater flexibility of NURBS compared to B-splines is illustrated in 2D. A quadrant of a circle cannot be represented using polynomial curves, but it can be represented as a NURBS curve of degree 2. The curve is $x^2 + y^2 = 1$, $x \geq 0$, $y \geq 0$. The general parameterization is

$$(x(u), y(u)) = \frac{w_0(1-u)^2(1, 0) + w_1 2u(1-u)(1, 1) + w_2 u^2(0, 1)}{w_0(1-u)^2 + w_1 2u(1-u) + w_2 u^2}$$

for $u \in [0, 1]$. The requirement that $x^2 + y^2 = 1$ leads to the weights constraint $2w_1^2 = w_0 w_2$. The choice of weights $w_0 = 1$, $w_1 = 1$, and $w_2 = 2$ leads to a well-known parameterization:

$$(x(u), y(u)) = \frac{(1-u^2, 2u)}{1+u^2}$$

If you were to tessellate the curve with an odd number of uniform samples of u , say, $u_i = i/(2n)$ for $0 \leq i \leq 2n$, then the resulting polyline is not symmetric about the midpoint $u = 1/2$. To obtain a symmetric tessellation, you need to choose $w_0 = w_2$. The weight constraint then implies $w_0 = w_1 \sqrt{2}$. The parameterization is then

$$(x(u), y(u)) = \frac{(\sqrt{2}(1-u)^2 + 2u(1-u), 2u(1-u) + \sqrt{2}u^2)}{\sqrt{2}(1-u)^2 + 2u(1-u) + \sqrt{2}u^2}$$

In either case we have a ratio of quadratic polynomials.

An algebraic construction of the same type, but quite a bit more tedious to solve, produces a ratio of quartic polynomials. The control points and control weights are required to be symmetric to obtain a tessellation that is symmetric about its midpoint. The middle weight is chosen as $w_2 = 4$ as shown:

$$(x(u), y(u)) = \frac{a_0(1, 0) + a_1(x_1, y_1) + a_2(x_2, x_2) + a_3(y_1, x_1) + a_4(0, 1)}{(1-u)^4 w_0 + 4(1-u)^3 u w_1 + 24(1-u)^2 u^2 + 4(1-u) u^3 w_1 + u^4 w_0}$$

where $a_0 = (1-u)^4 w_0$, $a_1 = 4(1-u)^3 u w_1$, $a_2 = 24(1-u)^2 u^2$, and $a_3 = 4(1-u) u^3 w_1$, $a_4 = u^4 w_0$. The parameters are $x_1 = 1$, $y_1 = (\sqrt{3} - 1)/\sqrt{3}$, $x_2 = (\sqrt{3} + 1)/(2\sqrt{3})$, $w_0 = 4\sqrt{3}(\sqrt{3} - 1)$, and $w_1 = 3/\sqrt{2}$. ■

We already have all the machinery in place to deal with the basis functions. The NURBS curve can be encapsulated in a class `NURBSCurve` that manages the control points `B[]`, the control weights `W[]`, and the degree `d`, and has a `BasisFunction` member `Nu`. The curve evaluator is

```
Point NURBSCurve::X (float u)
{
    int i = Nu.Compute(u);
    Point result = ZERO;
    float totalW = 0;
    for (int j = i - d; j <= i; j++)
    {
        float tmp = Nu.Basis(j) * W[j];
```

```

        result += tmp * B[j];
        totalW += tmp;
    }
    result /= totalW;
    return result;
}

```

For much more detail on B-spline and NURBS curves, see [Far90, Far99, CRE01, Rog01].

11.7 TENSION-CONTINUITY-BIAS SPLINES

Given an ordered list of points $\{\mathbf{P}_i\}_{i=0}^n$, the tension-continuity-bias (Kochonek-Bartel) splines provide a cubic interpolation between each pair $\mathbf{P}_i$ and $\mathbf{P}_{i+1}$ with varying properties specified at the endpoints [KB86]. These properties are *tension* τ , which controls how sharply the curve bends at a control point; *continuity* γ , which provides a smooth visual variation in the continuity at a control point ($\gamma = 0$ yields derivative continuity, but $\gamma \neq 0$ gives discontinuities); and *bias* β , which controls the direction of the path at a control point by taking a weighted combination of one-sided derivatives at that control point.

Using a Hermite interpolation basis $H_0(t) = 2t^3 - 3t^2 + 1$, $H_1(t) = -2t^3 + 3t^2$, $H_2(t) = t^3 - 2t^2 + t$, and $H_3(t) = t^3 - t^2$, a parametric cubic curve passing through points $\mathbf{P}_i$ and $\mathbf{P}_{i+1}$ with tangent vectors $\mathbf{T}_i$ and $\mathbf{T}_{i+1}$, respectively, is

$$\mathbf{X}_i(t) = H_0(t)\mathbf{P}_i + H_1(t)\mathbf{P}_{i+1} + H_2(t)\mathbf{T}_i + H_3(t)\mathbf{T}_{i+1} \quad (11.23)$$

where $0 \leq t \leq 1$. Catmull-Rom interpolation is a special case where $\mathbf{T}_i = (\mathbf{P}_{i+1} - \mathbf{P}_{i-1})/2$, a centered finite difference.

Equation (11.23) may be modified to allow specification of an *outgoing* tangent $\mathbf{T}_i^0$ at $t = 0$ and an *incoming* tangent $\mathbf{T}_{i+1}^1$ at $t = 1$:

$$\mathbf{x}_i(t) = H_0(t)\mathbf{P}_i + H_1(t)\mathbf{P}_{i+1} + H_2(t)\mathbf{T}_i^0 + H_3(t)\mathbf{T}_{i+1}^1 \quad (11.24)$$

Tension $\tau \in [-1, 1]$ can be introduced by using

$$\mathbf{T}_i^0 = \mathbf{T}_i^1 = \frac{(1 - \tau)}{2} ((\mathbf{P}_{i+1} - \mathbf{P}_i) + (\mathbf{P}_i - \mathbf{P}_{i-1}))$$

The Catmull-Rom spline occurs when $\tau = 0$. For τ near 1, the curve is *tightened* at the control point; τ near -1 produces *slack* at the control point. Varying τ changes the length of the tangent at the control point; a smaller tangent leads to a tightening, and a larger tangent leads to a slackening.

Continuity $\gamma \in [-1, 1]$ can be introduced by using

$$\mathbf{T}_i^0 = \left(\frac{1-\gamma}{2}(\mathbf{P}_{i+1} - \mathbf{P}_i) + \frac{1+\gamma}{2}(\mathbf{P}_i - \mathbf{P}_{i-1}) \right)$$

and

$$\mathbf{T}_i^1 = \left(\frac{1+\gamma}{2}(\mathbf{P}_{i+1} - \mathbf{P}_i) + \frac{1-\gamma}{2}(\mathbf{P}_i - \mathbf{P}_{i-1}) \right)$$

When $\gamma = 0$, the curve has a continuous tangent vector at the control point. As $|\gamma|$ increases, the resulting curve has a *corner* at the control point, the direction of the corner depending on the sign of γ .

Bias $\beta \in [-1, 1]$ can be introduced by using

$$\mathbf{T}_n^0 = \mathbf{T}_n^1 = \left(\frac{1-\beta}{2}(\mathbf{P}_{n+1} - \mathbf{P}_n) + \frac{1+\beta}{2}(\mathbf{P}_n - \mathbf{P}_{n-1}) \right)$$

When $\beta = 0$, the left and right one-sided tangents are equally weighted, producing the Catmull-Rom spline. For β near -1 , the outgoing tangent dominates the direction of the path of the curve through the control point—an effect referred to as *undershooting*. For β near 1 , the incoming tangent dominates—an effect referred to as *overshooting*.

The three effects may be combined into a single set of equations

$$\mathbf{T}_i^0 = \frac{(1-\tau)(1-\gamma)(1-\beta)}{2}(\mathbf{P}_{i+1} - \mathbf{P}_i) + \frac{(1-\tau)(1+\gamma)(1+\beta)}{2}(\mathbf{P}_i - \mathbf{P}_{i-1}) \quad (11.25)$$

and

$$\mathbf{T}_i^1 = \frac{(1-\tau)(1+\gamma)(1-\beta)}{2}(\mathbf{P}_{i+1} - \mathbf{P}_i) + \frac{(1-\tau)(1-\gamma)(1+\beta)}{2}(\mathbf{P}_i - \mathbf{P}_{i-1}) \quad (11.26)$$

These formulas assume a uniform spacing in time of the position samples. An adjustment can be made for nonuniform spacing. For Equation (11.25) the multiplier is $2\Delta_i/(\Delta_{i-1} + \Delta_i)$, and for Equation (11.26) the multiplier is $2\Delta_{i-1}/(\Delta_{i-1} + \Delta_i)$, where $\Delta_i = s_{i+1} - s_i$ and s_i is the sample time for position $\mathbf{P}_i$.

Figures 11.2 through 11.8 show a curve with six control points and various choices for tension, continuity, and bias at one of the control points.

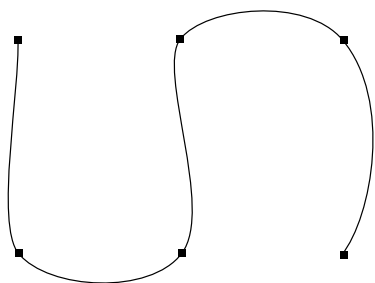


Figure 11.2 Parameters $\tau = 0, \gamma = 0, \beta = 0$.

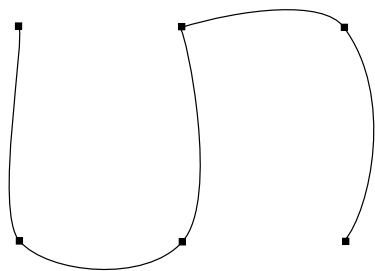


Figure 11.3 Parameters $\tau = 1, \gamma = 0, \beta = 0$.

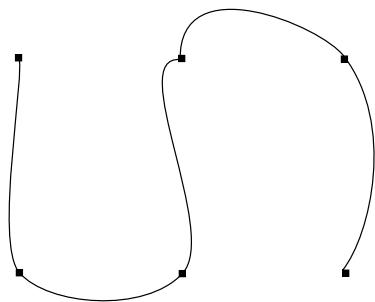


Figure 11.4 Parameters $\tau = 0, \gamma = 1, \beta = 0$.

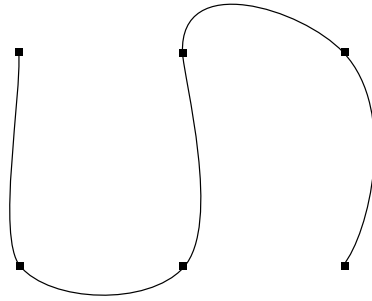


Figure 11.5 Parameters $\tau = 0$, $\gamma = 0$, $\beta = 1$.

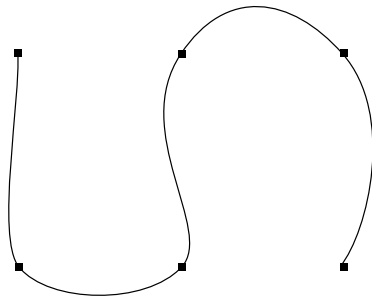


Figure 11.6 Parameters $\tau = -1$, $\gamma = 0$, $\beta = 0$.

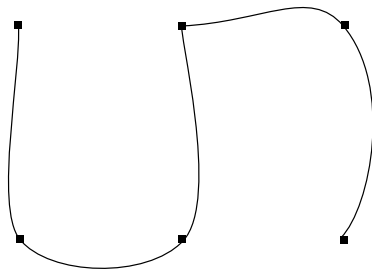
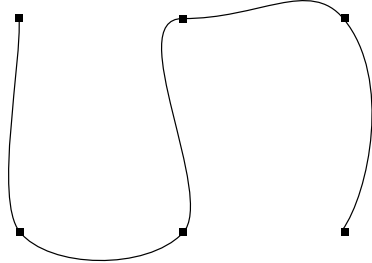


Figure 11.7 Parameters $\tau = 0$, $\gamma = -1$, $\beta = 0$.

Figure 11.8 Parameters $\tau = 0$, $\gamma = 0$, $\beta = -1$.

11.8 PARAMETRIC SUBDIVISION

For drawing purposes, it is sometimes necessary to produce a piecewise linear approximation to a curve with $n + 1$ curve points that will be the line segment endpoints. If t_i are the selected curve parameters for $0 \leq i \leq n$, then the set of points $\mathbf{X}_i = \mathbf{X}(t_i)$ for $0 \leq i \leq n$ is referred to as a *subdivision* of the curve. Three methods are discussed.

11.8.1 SUBDIVISION BY UNIFORM SAMPLING

The simplest way to subdivide is to uniformly sample $[t_{\min}, t_{\max}]$ as $t_i = t_{\min} + (t_{\max} - t_{\min})i/n$ for $0 \leq i \leq n$. Although easy to compute, the resulting polyline is not always a good approximation because places of large variation in the curve might be skipped. Figure 11.9 illustrates a uniform subdivision.

11.8.2 SUBDIVISION BY ARC LENGTH

This subdivision scheme selects a set of points that are equidistant from each other (measured with respect to arc length). Given $s_i = Li/n$, where L is the total curve length and $0 \leq i \leq n$, the algorithm for reparameterization by arc length can be applied to produce the corresponding t_i value. The subdivision points $\mathbf{X}(t_i)$ are then calculated. This method has the same problem as uniform sampling, namely, large variations of the curve over a small arc length may not be captured unless n is quite large. Figure 11.10 illustrates a subdivision by arc length.

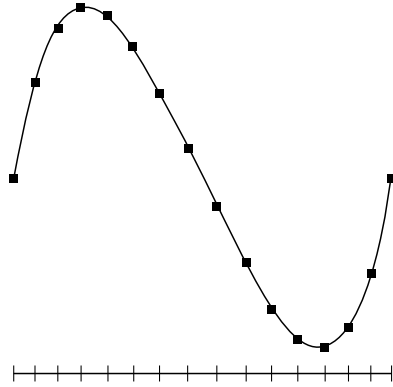


Figure 11.9 Uniform subdivision of a curve.

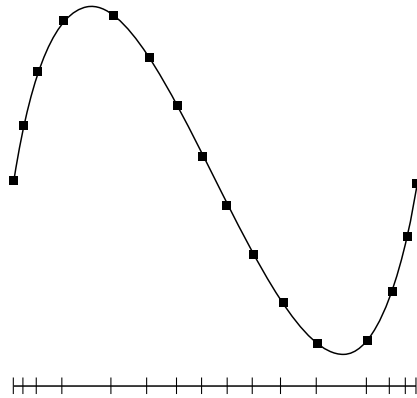


Figure 11.10 Subdivision of a curve by arc length.

11.8.3 SUBDIVISION BY MIDPOINT DISTANCE

This scheme produces a nonuniform sampling by recursively bisecting the parameter space. The bisection is actually performed, and the resulting curve point corresponding to the midpoint parameter is analyzed. If $\mathbf{A}$ and $\mathbf{B}$ are the endpoints of the segment and if $\mathbf{C}$ is the computed point in the bisection step, then the distance D_0 from $\mathbf{C}$ to the segment is computed. If $D_1 = |\mathbf{B} - \mathbf{A}|$, then $\mathbf{C}$ is added to the tessella-

tion if $D_0/D_1 > \varepsilon$ for an application-specified maximum relative error of $\varepsilon > 0$. The pseudocode is given next. Rather than maintaining a doubly linked list to handle the insertion of points on subdivision, the code maintains a singly linked list of ordered points.

```

Input: Curve  $x(t)$  with  $t$  in  $[t_{\min}, t_{\max}]$ 
       $m$ , the maximum level of subdivision
       $\varepsilon$ , the maximum relative error
      subdivision {}, an empty list
Output:  $n \geq 1$  and subdivision  $\{p[0], \dots, p[n]\}$ 

bool Bisect (int level, float t0, Point x0, float t1, Point x1)
{
    if (level > 0)
    {
        tm = (t0 + t1)/2;
        xm = x(tm);
        d0 = length of segment  $\langle x0, x1 \rangle$ 
        d1 = distance from xm to segment  $\langle x0, x1 \rangle$ ;

        if (d1/d0 > epsilon)
        {
            Bisect(level - 1, t0, x0, tm, xm);
            Bisect(level - 1, tm, xm, t1, x1);
            return;
        }
    }

    add x1 to end of list;
}

Initial call:
    subdivision = {  $x(t_{\min})$  };
    Bisect(m, tmin, x(tmin), tmax, x(tmax));

```

Figure 11.11 illustrates a subdivision using this method.

11.8.4 FAST SUBDIVISION FOR CUBIC CURVES

Cubic polynomial curves can be subdivided by using a heuristic of flatness to stop the subdivisions. The method described in this section uses the magnitude of the second-derivative vector of the curve multiplied by the length of a subinterval as an estimate of how flat (or curved) the curve is on that subinterval. If the magnitude is smaller

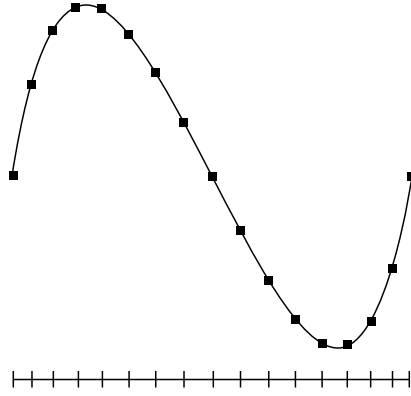


Figure 11.11 Subdivision of a curve by midpoint distance.

than the application-specified tolerance, the subdivision step is not executed. The classic case where the heuristic fails is an S-shaped curve whose point of inflection is the midpoint of the given interval. The second derivative is zero at the midpoint, so the subdivision step is not executed. However, the curve can have significant variation from the line segment connecting the endpoints of the interval. It is simple enough to trap this case and subdivide once to get past the inflection.

The subdivision method is naturally recursive. We take advantage of this fact and use a central differencing scheme to compute the curve points [WW92].

Let the cubic curve be $\mathbf{X}(t) = \sum_{i=0}^3 \mathbf{C}_i t^i$ for $t \in [t_{\min}, t_{\max}]$. Using a Taylor series to represent the curve, the following equations can be derived:

$$\mathbf{X}(t \pm \delta) = \mathbf{X}(t) \pm \delta \mathbf{X}'(t) + \frac{1}{2} \delta^2 \mathbf{X}''(t) \pm \frac{1}{6} \delta^3 \mathbf{X}'''(t)$$

There are no additional terms in the Taylor series since the curve is a cubic polynomial. Taking the average of the two equations and solving for $\mathbf{X}(t)$ yields

$$\mathbf{X}(t) = \frac{1}{2} \left(\mathbf{X}(t + \delta) + \mathbf{X}(t - \delta) - \delta^2 \mathbf{X}''(t) \right) \quad (11.27)$$

Expanding the second-derivative term as a Taylor series, we obtain

$$\mathbf{X}''(t \pm \delta) = \mathbf{X}''(t) \pm \delta \mathbf{X}'''(t)$$

Adding these and solving for $\mathbf{X}''(t)$ yields

$$\mathbf{X}''(t) = \frac{1}{2} \left(\mathbf{X}''(t + \delta) + \mathbf{X}''(t - \delta) \right) \quad (11.28)$$

Equation (11.28) allows us to compute the second derivative of the curve at the midpoint t of the interval $[t - \delta, t + \delta]$. This can be substituted in Equation (11.27) to compute the curve at the midpoint.

The pseudocode for the recursive subdivision is

```

Input: Cubic curve  $x(t)$  with  $t$  in  $[t_{\min}, t_{\max}]$ 
       epsilon, the maximum relative error (units of squared length)
       subdivision {}, an empty list
Output:  $n \geq 1$  and subdivision  $\{p[0], \dots, p[n]\}$ 

void Subdivide (float t0, float t1, Point x0, Point x1, Point sd0, Point sd1)
{
    // x0 and x1 are endpoints.
    // sd0 and sd1 are second derivatives at the endpoints.

    sdmid = 0.5 * (sd0 + sd1);
    d = t1 - t0;
    dsqr = d * d;
    nonlinearity = dsqr * sdmid;
    if (SquaredLength(nonlinearity) > epsilon)
    {
        tmid = 0.5 * (t0 + t1);
        xmid = 0.5 * (x0 + x1 - nonlinearity);
        insert xmid in subdivision between x0 and x1;
        Subdivide(t0, tmid, x0, xmid, sd0, sdmid);
        Subdivide(tmid, t1, xmid, x1, sdmid, sd1);
    }
}

Initial call:
x0 = x(tmin);
x1 = x(tmax);
sd0 = x''(tmin);
sd1 = x''(tmax);
subdivision = { x0, x1 };
Subdivide(tmin, tmax, x0, x1, sd0, sd1);

```

11.9 ORIENTATION OF OBJECTS ON CURVED PATHS

Specifying a path of motion for an object, including the orientation of the object at each point along the path, is called *path controlling*. For example, if a model airplane is given a path to follow, the orientation of the model airplane along the path should be representative of the real thing. If the path takes the airplane to the right, the plane should change orientation and bank to the right.

Let the specified path of the object be a curve $\mathbf{X}(t)$ for some domain of t values. The orientation can be specified as a rotation matrix $R(t)$, where the columns of R are the coordinate axes at each point on the path. The columns are ordered in the following sense. The first column represents a *direction* vector, but is not required by the theory to be the tangent to the curve. The second column is an *up* vector, and the third column is a *right* vector. There are many ways to specify orientation, but the two most common are to use either the Frenet frame of the curve or a coordinate system with a fixed up vector where the upward direction is specific to an application.

11.9.1 ORIENTATION USING THE FRENET FRAME

This method requires that $\mathbf{X}(t)$ be twice differentiable so that the normal vector is well defined. Recall from the curve definitions that the Frenet frame consists of a tangent vector $\mathbf{T}(t)$, a normal vector $\mathbf{N}(t)$, and a binormal vector $\mathbf{B}(t) = \mathbf{T} \times \mathbf{N}$. The tangent vector is a unit-length vector with direction $\mathbf{X}'(t)$. The normal vector represents a force parallel to the acceleration of the object. The orientation matrix is $R(t) = [\mathbf{T}(t) \ \mathbf{N}(t) \ \mathbf{B}(t)]$.

The drawback to using a Frenet frame occurs when you pass through a point of inflection. The normal vector is discontinuous in time, and the frame can flip on you.

11.9.2 ORIENTATION USING A FIXED UP-VECTOR

This method requires that $\mathbf{X}(t)$ be once differentiable so that the tangent vector is well defined. An application must specify a vector $\mathbf{U}$ that points in the upward direction. The tangent vector is $\mathbf{T}(t)$, the unit-length vector with direction $\mathbf{X}'(t)$:

$$\mathbf{T}(t) = \frac{\mathbf{X}'(t)}{|\mathbf{X}'(t)|}$$

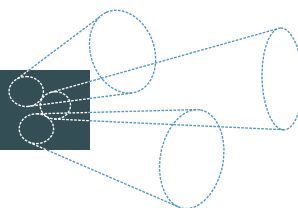
The first column of $R(t)$ is chosen to be $\mathbf{T}$. The third column of $R(t)$, $\mathbf{B}(t)$, is computed as the unit-length cross product between $\mathbf{T}$ and $\mathbf{U}$:

$$\mathbf{B}(t) = \frac{\mathbf{T}(t) \times \mathbf{U}}{|\mathbf{T}(t) \times \mathbf{U}|}$$

The second column of $R(t)$, $\mathbf{N}(t)$, is chosen as

$$\mathbf{N}(t) = \mathbf{B}(t) \times \mathbf{T}(t)$$

The only item of concern in using this method for orientation is that $\mathbf{T}(t)$ should never be parallel to $\mathbf{U}$, otherwise $\mathbf{B}(t) = \mathbf{0}$ and the coordinate system cannot be constructed. For numerical reasons, it is better to constrain the curve so that the angle between $\mathbf{T}$ and $\mathbf{U}$ is larger than a predefined positive minimum angle.



SURFACES

When I wrote the first edition of this book, I predicted that the powerful processors and the limited memory on the game consoles would steer developers and artists toward using control-point surfaces. The premise was that only a small number of control points need to be stored in memory, but the processor can tessellate rapidly to any reasonable level of subdivision. It appears that my prediction was premature. Game artists appear to remain comfortable with polygonal models, and each generation of console tends to have a lot more memory than the previous, so the urgency is lacking to produce high-resolution tessellations from a small amount of data.

That said, surfaces are still a reasonable way of generating smooth and complex shapes, even if the end result is a polygonal tessellation that is created from the surface and exported from the modeling package as a triangle mesh. This section covers the basic definitions for surfaces, as well as examples of a few special types of surfaces.

12.1 INTRODUCTION

A *parametric surface patch* is a function $\mathbf{X} : [u_{\min}, u_{\max}] \times [v_{\min}, v_{\max}] \subset \mathbb{R}^2 \rightarrow \mathbb{R}^3$. The surface boundary curves are $\mathbf{X}(u_{\min}, v)$, $\mathbf{x}(u_{\max}, v)$, $\mathbf{X}(u, v_{\min})$, and $\mathbf{X}(u, v_{\max})$. *Tangent vectors* to the surface are the partial derivatives $\mathbf{X}_u = \partial \mathbf{X} / \partial u$ and $\mathbf{X}_v = \partial \mathbf{X} / \partial v$. A *normal vector* at each point on the surface is the cross product of the partial derivatives, $\mathbf{N} = \mathbf{X}_u \times \mathbf{X}_v$. If unit-length normals are required, the cross product can be normalized. The patch is actually called a *rectangle patch* because the domain is a rectangular set in the parameter space. The standard domain for rectangle patches is

$[0, 1]^2$. Another common type of patch is a *triangle patch*, where the domain is a triangular set. The standard domain for a triangle patch is $u \geq 0$, $v \geq 0$, and $u + v \leq 1$. Generally, the parametric domain can be a set $D \subseteq \mathbb{R}^2$.

An *implicit surface* is defined by level sets $F(\mathbf{X}) = c$ for a function $F : \mathbb{R}^3 \rightarrow \mathbb{R}$. A normal vector at each point is the gradient, $\mathbf{N} = \nabla F$. If unit-length normals are required, the gradient can be normalized. Two linearly independent tangent vectors $\mathbf{U}$ and $\mathbf{V}$ can be constructed from $\mathbf{N}$. A reasonable algorithm for constructing the tangents is

```
Vector3 N = (x,y,z); // unit-length normal
Vector3 U, V; // unit-length tangents
if (|x| >= |y| and |x| >= |z|)
{
    U = (y,-x,0)/sqrt(x * x + y * y);
}
else
{
    U = (0,y,-z)/sqrt(y * y + z * z);
}
V = Cross(N,U);
```

The typical implicit surfaces you encounter are *quadric surfaces*, where $F(x, y, z)$ is a quadratic function of its inputs.

A surface that is the *graph* of a function f can be described either parametrically as $(x, y, f(x, y))$ or implicitly as $F(x, y, z) = z - f(x, y) = 0$. In the first case, two tangents are $\mathbf{U} = (1, 0, \partial f / \partial x)$ and $\mathbf{V} = (0, 1, \partial f / \partial y)$, and a normal is the cross product $\mathbf{N} = (-\partial f / \partial x, -\partial f / \partial y, 1) = (-\nabla f, 1)$. In the second case, note that $\nabla F = \mathbf{N}$.

12.2 BÉZIER RECTANGLE PATCHES

Bézier rectangle patches are popular with game programmers for their mathematical simplicity and ease of use.

12.2.1 DEFINITIONS

Given a rectangular lattice of 3D control points $\mathbf{P}_{i_0, i_1}$ for $0 \leq i_0 \leq n_0$ and $0 \leq i_1 \leq n_1$, the Bézier rectangle patch for the points is

$$\mathbf{X}(s, t) = \sum_{i_0=0}^{n_0} \sum_{i_1=0}^{n_1} B_{n_0, i_0}(s) B_{n_1, i_1}(t) \mathbf{P}_{i_0, i_1}$$

for $(s, t) \in [0, 1]^2$ and where the coefficients are products of the Bernstein polynomials defined in Equation (11.1). The first-order partial derivatives of the patch are

$$\mathbf{X}_s(s, t) = n_0 \sum_{i_0=0}^{n_0-1} \sum_{i_1=0}^{n_1} B_{n_0-1, i_0}(s) B_{n_1, i_1}(t) (\mathbf{P}_{i_0+1, i_1} - \mathbf{P}_{i_0, i_1})$$

and

$$\mathbf{X}_t(s, t) = n_1 \sum_{i_0=0}^{n_0} \sum_{i_1=0}^{n_1-1} B_{n_0, i_0}(s) B_{n_1-1, i_1}(t) (\mathbf{P}_{i_0, i_1+1} - \mathbf{P}_{i_0, i_1})$$

12.2.2 EVALUATION

As for Bézier curves, the same trade-off of speed versus accuracy must be made for Bézier rectangles. The choice here is for speed. Each Bernstein polynomial is computed, and then the product of the polynomials for each term is computed. There are $n_0 + 1$ evaluations of $B_{n_0, i_0}(s)$, $n_1 + 1$ evaluations of $B_{n_1, i_1}(t)$, and $n_0 n_1$ multiplications for pairs of the evaluated polynomials.

The de Casteljau algorithm repeatedly computes convex combinations and is generally more stable, but it uses more floating-point operations. For example, let's compare it to the Bernstein form of evaluation for bilinear interpolation, the case where $n_0 = 1$ and $n_1 = 1$. The Bernstein form for evaluation is

$$(1-s)(1-t)\mathbf{P}_{0,0} + (1-s)t\mathbf{P}_{0,1} + s(1-t)\mathbf{P}_{1,0} + st\mathbf{P}_{1,1}$$

and requires two subtractions, nine additions, and 16 multiplications. The de Casteljau form for evaluation is

$$(1-s)((1-t)\mathbf{P}_{0,0} + t\mathbf{P}_{0,1}) + s((1-t)\mathbf{P}_{1,0} + t\mathbf{P}_{1,1})$$

and requires two subtractions, nine additions, and 18 multiplications.

12.2.3 DEGREE ELEVATION

A Bézier rectangle patch of degree (n_0, n_1) can be written as a patch of degree $(n_0 + 1, n_1)$. The process is similar to that of a Bézier curve where the equation is multiplied by $1 = (1-s) + s$ and formally expanded. The degree-elevated patch is

$$\mathbf{X}(s, t) = \sum_{i_0=0}^{n_0+1} \sum_{i_1=0}^{n_1} B_{n_0+1, i_0}(s) B_{n_1, i_1}(t) \left[\left(1 - \frac{i_0}{n_0+1}\right) \mathbf{P}_{i_0, i_1} + \frac{i_0}{n_0+1} \mathbf{P}_{i_0-1, i_1} \right]$$

The patch can be similarly degree elevated to one of degree $(n_0, n_1 + 1)$:

$$\mathbf{X}(s, t) = \sum_{i_0=0}^{n_0} \sum_{i_1=0}^{n_1+1} B_{n_0, i_0}(s) B_{n_1+1, i_1}(t) \left[\left(1 - \frac{i_1}{n_1+1} \right) \mathbf{P}_{i_0, i_1} + \frac{i_1}{n_1+1} \mathbf{P}_{i_0, i_1-1} \right]$$

The patch can be degree elevated in both components to one of degree $(n_0 + 1, n_1 + 1)$:

$$\mathbf{X}(s, t) = \sum_{i_0=0}^{n_0+1} \sum_{i_1=0}^{n_1+1} B_{n_0+1, i_0}(s) B_{n_1+1, i_1}(t) \mathbf{Q}_{i_0, i_1}$$

where

$$\mathbf{Q}_{i_0, i_1} = \begin{bmatrix} 1 - \frac{i_0}{n_0+1} & \frac{i_0}{n_0+1} \end{bmatrix} \begin{bmatrix} \mathbf{P}_{i_0, i_1} & \mathbf{P}_{i_0, i_1-1} \\ \mathbf{P}_{i_0-1, i_1} & \mathbf{P}_{i_0-1, i_1-1} \end{bmatrix} \begin{bmatrix} 1 - \frac{i_1}{n_1+1} \\ \frac{i_1}{n_1+1} \end{bmatrix}$$

The right-hand side is evaluated symbolically as a product of the three matrices.

12.2.4 DEGREE REDUCTION

A Bézier rectangle patch can be reduced in degree with similar constraints as in the Bézier curve case. The reduced patch in almost all cases is an approximation to the original patch. A least-squares fit is used to obtain the reduced patch.

Let the original surface be

$$\mathbf{X}(s, t) = \sum_{i_0=0}^{n_0} \sum_{i_1=0}^{n_1} B_{n_0, i_0}(s) B_{n_1, i_1}(t) \mathbf{P}_{i_0, i_1}$$

and let the degree-reduced surface be

$$\mathbf{Y}(s, t) = \sum_{i_0=0}^{m_0} \sum_{i_1=0}^{m_1} B_{m_0, i_0}(s) B_{m_1, i_1}(t) \mathbf{Q}_{i_0, i_1}$$

where $m_0 \leq n_0$ and $m_1 \leq n_1$. For degree reduction of Bézier curves, we imposed the constraint that the end points of the two curves be the same. The extension to rectangle patches is to require that the four corner points match between the two patches. Although it is possible to apply a least-squares fit to construct the remaining control points, a better approach looks ahead to the situations where two patches have a common boundary curve. The reduction scheme when applied to the two adjacent patches should guarantee that the patches match on the common reduced boundary curve. The algorithm for a single patch should therefore degree-reduce the four boundary curves first, then compute the remaining interior control points

using the least-squares fit. The to-be-determined interior points are $\mathbf{Q}_{i_0, i_1}$ for $1 \leq i_0 \leq m_0 - 1$ and $1 \leq i_1 \leq m_1 - 1$. These are chosen to minimize the integral of the squared differences of the two surfaces:

$$E(\mathbf{Q}_{1,1}, \dots, \mathbf{Q}_{m_0-1, m_1-1}) = \int_0^1 \int_0^1 |\mathbf{X}(s, t) - \mathbf{Y}(s, t)|^2 ds dt$$

The values of the interior control points are determined by setting all the partial derivatives of E to zero, $\partial E / \partial \mathbf{Q}_{i_0, i_1} = 0$ for $1 \leq i_0 \leq m_0 - 1$ and $1 \leq i_1 \leq m_1 - 1$. This leads to $(m_0 - 1)(m_1 - 1)$ equations in the same number of unknown control points:

$$\begin{aligned} & \sum_{i_0=0}^{m_0} \sum_{i_1=0}^{m_1} \frac{(2m_0 + 1)(2m_1 + 1)C(m_0; i_0)C(m_1; i_1)}{C(2m_0; i_0 + j_0)C(2m_1; i_1 + j_1)} \mathbf{Q}_{i_0, i_1} \\ &= \sum_{i_0=0}^{n_0} \sum_{i_1=0}^{n_1} \frac{(n_0 + m_0 + 1)(n_1 + m_1 + 1)C(n_0; i_0)C(n_1; i_1)}{C(n_0 + m_0; i_0 + j_0)C(n_1 + m_1; i_1 + j_1)} \mathbf{P}_{i_0, i_1} \end{aligned}$$

The system always has a solution.

For example, solving the equations symbolically for degree reduction of a bicubic patch to a biquadratic patch, $n_0 = n_1 = 3$ and $m_0 = m_1 = 2$:

$$\mathbf{Q}_{0,0} = \mathbf{P}_{0,0}$$

$$\mathbf{Q}_{0,2} = \mathbf{P}_{0,3}$$

$$\mathbf{Q}_{2,0} = \mathbf{P}_{3,0}$$

$$\mathbf{Q}_{2,2} = \mathbf{P}_{3,3}$$

$$\mathbf{Q}_{0,1} = \frac{1}{4} (-\mathbf{P}_{0,0} + 3\mathbf{P}_{0,1} + 3\mathbf{P}_{0,2} - \mathbf{P}_{0,3})$$

$$\mathbf{Q}_{1,0} = \frac{1}{4} (-\mathbf{P}_{0,0} + 3\mathbf{P}_{1,0} + 3\mathbf{P}_{2,0} - \mathbf{P}_{3,0})$$

$$\mathbf{Q}_{1,2} = \frac{1}{4} (-\mathbf{P}_{0,3} + 3\mathbf{P}_{1,3} + 3\mathbf{P}_{2,3} - \mathbf{P}_{3,3})$$

$$\mathbf{Q}_{2,1} = \frac{1}{4} (-\mathbf{P}_{3,0} + 3\mathbf{P}_{3,1} + 3\mathbf{P}_{3,2} - \mathbf{P}_{3,3})$$

$$\begin{aligned} \mathbf{Q}_{1,1} = & \frac{1}{16} (\mathbf{P}_{0,0} - 3\mathbf{P}_{0,1} - 3\mathbf{P}_{0,2} + \mathbf{P}_{0,3} - 3\mathbf{P}_{1,0} + 9\mathbf{P}_{1,1} + 9\mathbf{P}_{1,2} - 3\mathbf{P}_{1,3} \\ & - 3\mathbf{P}_{2,0} + 9\mathbf{P}_{2,1} + 9\mathbf{P}_{2,2} - 3\mathbf{P}_{2,3} + \mathbf{P}_{3,0} - 3\mathbf{P}_{3,1} - 3\mathbf{P}_{3,2} + \mathbf{P}_{3,3}) \end{aligned}$$

v^2			0			$2v$		
$2vw$	$2uv$		$-2v$	$2v$		$2(-v + w)$	$2u$	
w^2	$2uw$	u^2	$-2w$	$2(-u + w)$	$2u$	$-2w$	$-2u$	0
Coefficients of $\mathbf{x}$			Coefficients of $\mathbf{x}_u$			Coefficients of $\mathbf{x}_v$		

Figure 12.1 Polynomial coefficients for $n = 2$.

12.3 BÉZIER TRIANGLE PATCHES

Bézier triangle patches are slightly more complicated to use than Bézier rectangle patches, but they are useful for creating models of arbitrary complexity.

12.3.1 DEFINITIONS

Given a triangle lattice of 3D control points $\mathbf{P}_{i_0, i_1, i_2}$ for $i_0 \geq 0$, $i_1 \geq 0$, $i_2 \geq 0$, and $i_0 + i_1 + i_2 = n$, the Bézier triangle patch for the points is

$$\mathbf{X}(u, v, w) = \sum_{|I|=n} B_{n,I}(u, v, w) \mathbf{P}_I$$

where $I = (i_0, i_1, i_2)$, $|I| = i_0 + i_1 + i_2$, $u \geq 0$, $v \geq 0$, $w \geq 0$, and $u + v + w = 1$. The summation involves $(n+1)(n+2)/2$ terms. The Bernstein polynomial coefficients are

$$B_{n,I}(u, v, w) = C(n; i_0, i_1, i_2) u^{i_0} v^{i_1} w^{i_2} = \frac{n!}{i_0! i_1! i_2!} u^{i_0} v^{i_1} w^{i_2}$$

The first-order partial derivatives $\mathbf{X}_u$ and $\mathbf{X}_v$ can be computed with respect to u or v , where $w = 1 - u - v$. While the symbolic formula can be computed from the equation for $\mathbf{X}(u, v, w)$, it is simpler to visualize the coefficients for $\mathbf{X}$, $\mathbf{X}_u$, and $\mathbf{X}_v$ as triangles of terms. The multi-index $I = (i_0, i_1, i_2)$ varies as follows. The index i_0 increases from left to right, the index i_1 varies from bottom to top, and the index $i_2 = n - i_0 - i_1$. Figures 12.1, 12.2, and 12.3 show the coefficient triangles for the cases $n = 2$, $n = 3$, and $n = 4$, respectively.

12.3.2 EVALUATION

Evaluation of $\mathbf{x}$ or its derivatives is a matter of computing the coefficients, illustrated in Figures 12.1 through 12.3, multiplying them times the control points, and summing. In an implementation, the triangle coefficients are stored in a 1D array. The

v^3				0			
$3v^2w$	$3uv^2$			$-3v^2$	$3v^2$		
$3vw^2$	$6uvw$	$3u^2v$		$-6vw$	$6v(-u + w)$	$6uv$	
w^3	$3uw^2$	$3u^2w$	u^3	$-3w^2$	$3w(-2u + w)$	$3u(-u + 2w)$	$3u^2$

Coefficients of $\mathbf{x}$
Coefficients of $\mathbf{x}_u$

$3v^2$			
$3v(-v + 2w)$	$6uv$		
$3w(-2v + w)$	$6u(-v + w)$	$3u^2$	
$-3w^2$	$-6uw$	$-3u^2$	0

Coefficients of $\mathbf{x}_v$

Figure 12.2 Polynomial coefficients for $n = 3$.

v^4				
$4v^3w$	$4uv^3$			
$6v^2w^2$	$12uv^2w$	$6u^2v^2$		
$4vw^3$	$12uvw^2$	$12u^2vw$	$4u^3v$	
w^4	$4uw^3$	$6u^2w^2$	$4u^3w$	u^4

Coefficients of $\mathbf{x}$

0				
$-4v^3$	$4v^3$			
$-12v^2w$	$12v^2(-u + w)$	$12uv^2$		
$-12vw^2$	$12vw(-2u + w)$	$12uv(-u + 2w)$	$12u^2v$	
$-4w^3$	$4w^2(-3u + w)$	$12uw(-u + w)$	$4u^2(-u + 3w)$	$4u^3$

Coefficients of $\mathbf{x}_u$

$4v^3$				
$4v^2(-v + 3w)$	$12uv^2$			
$12vw(-v + w)$	$12uv(-v + 2w)$	$12u^2v$		
$4w^2(-3v + w)$	$12uw(-2v + w)$	$12u^2(-v + w)$	$4u^3$	
$-4w^3$	$-12uw^2$	$-12u^2w$	$-4u^2w$	$-4u^3$

Coefficients of $\mathbf{x}_v$

Figure 12.3 Polynomial coefficients for $n = 4$.

rows of the coefficient tables are stored bottom first ($n + 1$ items, scanned left to right) through top last (1 item). The coefficients themselves are computed to minimize arithmetic operations by saving intermediate products and sums.

12.3.3 DEGREE ELEVATION

A Bézier triangle patch of degree n can be written as a patch of degree $n + 1$. The idea is to formally multiply the original patch by $1 = u + v + w$ so that the surface does not change, but the degree does. The degree-elevated patch is defined by

$$\mathbf{X}(u, v, w) = (u + v + w) \sum_{|I|=n} B_{n,I}(u, v, w) \mathbf{P}_I = \sum_{|I|=n+1} B_{n+1,I}(u, v, w) \mathbf{Q}_I$$

where the degree-elevated control points are

$$\mathbf{Q}_I = \frac{1}{n+1} \left(i_0 \mathbf{P}_{i_0-1, i_1, i_2} + i_1 \mathbf{P}_{i_0, i_1-1, i_2} + i_2 \mathbf{P}_{i_0, i_1, i_2-1} \right)$$

12.3.4 DEGREE REDUCTION

A Bézier triangle patch can be reduced in degree with the same constraints as for Bézier curves. The reduced patch in almost all cases is an approximation to the original patch. A least-squares fit can be used to obtain the reduction.

Let the original surface be $\mathbf{X}(u, v, w) = \sum_{|I|=n} B_{n,I}(u, v, w) \mathbf{P}_I$, and let the degree-reduced surface be $\mathbf{Y}(u, v, w) = \sum_{|I|=m} B_{m,I}(u, v, w) \mathbf{Q}_I$, where $m < n$. For Bézier curves, we imposed the constraint that the endpoints of the two curves must match. The extension to triangle patches is to require that the three corner points match between the two patches. Just as with rectangle patches, the boundary curves of the patch are reduced separately, and the interior points of the patch are determined from a surface least-squares fit. This guarantees that applying a reduction in degree across multiple patches with shared boundaries will maintain continuity across those boundaries. The interior points are chosen to minimize the integral of the squared differences of the two patches:

$$E(\cdot) = \int_0^1 \int_0^{1-v} |\mathbf{X}(u, v, 1-u-v) - \mathbf{Y}(u, v, 1-u-v)|^2 du dv$$

The arguments for $E(\cdot)$ are the interior control points for the approximating patch.

The values of the interior control points are determined by setting all the partial derivatives of E to zero, $\partial E / \partial \mathbf{Q}_J$ for those indices $J = (j_0, j_1, j_2)$ with $j_0 j_1 j_2 \neq 0$. This leads to the equation

$$\sum_{|I|=m} a_{JI} \mathbf{Q}_I = \sum_{|I|=n} b_{JI} \mathbf{P}_I$$

where

$$a_{JI} = \frac{C(m; J)C(m; I)}{(2m+2)(2m+1)C(2m; I+J)}$$

and

$$b_{JI} = \frac{C(m; J)C(n; I)}{(n+m+2)(n+m+1)C(n+m; I+J)}$$

The system always has a solution.

The equations can be solved symbolically for some cases of interest. For $n = 4$ and $m = 3$, the solution is

$$\mathbf{Q}_{300} = \mathbf{P}_{400}$$

$$\mathbf{Q}_{030} = \mathbf{P}_{040}$$

$$\mathbf{Q}_{003} = \mathbf{P}_{004}$$

$$\mathbf{Q}_{012} = \frac{1}{42} (-11\mathbf{P}_{004} + 44\mathbf{P}_{013} + 18\mathbf{P}_{022} - 12\mathbf{P}_{031} + 3\mathbf{P}_{040})$$

$$\mathbf{Q}_{021} = \frac{1}{42} (3\mathbf{P}_{004} - 12\mathbf{P}_{013} + 18\mathbf{P}_{022} + 44\mathbf{P}_{031} - 11\mathbf{P}_{040})$$

$$\mathbf{Q}_{102} = \frac{1}{42} (-11\mathbf{P}_{004} + 44\mathbf{P}_{103} + 18\mathbf{P}_{202} - 12\mathbf{P}_{301} + 3\mathbf{P}_{400})$$

$$\mathbf{Q}_{201} = \frac{1}{42} (3\mathbf{P}_{004} - 12\mathbf{P}_{103} + 18\mathbf{P}_{202} + 44\mathbf{P}_{301} - 11\mathbf{P}_{400})$$

$$\mathbf{Q}_{210} = \frac{1}{42} (-11\mathbf{P}_{400} + 44\mathbf{P}_{310} + 18\mathbf{P}_{220} - 12\mathbf{P}_{130} + 3\mathbf{P}_{040})$$

$$\mathbf{Q}_{120} = \frac{1}{42} (3\mathbf{P}_{400} - 12\mathbf{P}_{310} + 18\mathbf{P}_{220} + 44\mathbf{P}_{130} - 11\mathbf{P}_{040})$$

$$\mathbf{Q}_{111} = \frac{1}{2520} (5(\mathbf{P}_{004} + \mathbf{P}_{040} + \mathbf{P}_{400}) + 8(\mathbf{P}_{103} + \mathbf{P}_{301} + \mathbf{P}_{013} + \mathbf{P}_{310} + \mathbf{P}_{031} + \mathbf{P}_{130})$$

$$+ 9(\mathbf{P}_{202} + \mathbf{P}_{022} + \mathbf{P}_{220}) + 12(\mathbf{P}_{112} + \mathbf{P}_{211} + \mathbf{P}_{121}))$$

$$- \frac{2}{560} (\mathbf{Q}_{300} + \mathbf{Q}_{030} + \mathbf{Q}_{003}) - \frac{3}{560} (\mathbf{Q}_{012} + \mathbf{Q}_{021} + \mathbf{Q}_{102} + \mathbf{Q}_{201} + \mathbf{Q}_{120} + \mathbf{Q}_{210})$$

For $n = 4$ and $m = 2$, the solution is

$$\mathbf{Q}_{200} = \mathbf{P}_{400}$$

$$\mathbf{Q}_{020} = \mathbf{P}_{040}$$

$$\mathbf{Q}_{002} = \mathbf{P}_{004}$$

$$\mathbf{Q}_{101} = \frac{1}{28} (-11\mathbf{P}_{004} + 16\mathbf{P}_{103} + 18\mathbf{P}_{202} + 16\mathbf{P}_{301} - 11\mathbf{P}_{400})$$

$$\mathbf{Q}_{011} = \frac{1}{28} (-11\mathbf{P}_{004} + 16\mathbf{P}_{013} + 18\mathbf{P}_{022} + 16\mathbf{P}_{031} - 11\mathbf{P}_{040})$$

$$\mathbf{Q}_{110} = \frac{1}{28} (-11\mathbf{P}_{400} + 16\mathbf{P}_{310} + 18\mathbf{P}_{220} + 16\mathbf{P}_{130} - 11\mathbf{P}_{040})$$

For $n = 3$ and $m = 2$, the solution is

$$\mathbf{Q}_{200} = \mathbf{P}_{300}$$

$$\mathbf{Q}_{020} = \mathbf{P}_{030}$$

$$\mathbf{Q}_{002} = \mathbf{P}_{003}$$

$$\mathbf{Q}_{101} = \frac{1}{4} (-\mathbf{P}_{003} + 3\mathbf{P}_{102} + 3\mathbf{P}_{201} - \mathbf{P}_{300})$$

$$\mathbf{Q}_{011} = \frac{1}{4} (-\mathbf{P}_{003} + 3\mathbf{P}_{012} + 3\mathbf{P}_{021} - \mathbf{P}_{030})$$

$$\mathbf{Q}_{110} = \frac{1}{4} (-\mathbf{P}_{030} + 3\mathbf{P}_{210} + 3\mathbf{P}_{120} - \mathbf{P}_{030})$$

12.4 B-SPLINE RECTANGLE PATCHES

The simplest extension of the concept of B-spline curves to surfaces is to blend a rectangular array of control points $\mathbf{P}_{i_0, i_1}$ for $0 \leq i_0 \leq n_0$ and $0 \leq i_1 \leq n_1$. The blending occurs separately in each dimension, leading to a rectangle surface patch. The degree must be specified for each dimension, d_0 and d_1 , with $1 \leq d_i \leq n_i$. The surface patch is defined by

$$\mathbf{X}(u, v) = \sum_{i_0=0}^{n_0} \sum_{i_1=0}^{n_1} N_{i_0, d_0}(u) N_{i_1, d_1}(v) \mathbf{P}_{i_0, i_1} \quad (12.1)$$

We already have the mechanism in place for computing the basis functions, namely, from B-spline curves. The B-spline surface is encapsulated in a class `BSplineSurface` and manages the control points `P[][]`, the degrees `d0` and `d1`, and has `BasisFunction` objects `Nu` and `Nv`. The surface evaluation is

```

Point BSplineSurface::X (float u, float v)
{
    int i0 = Nu.Compute(u), i1 = Nv.Compute(v);
    Point result = ZERO;
    for (int j0 = i0 - d0; j0 <= i0; j0++)
    {
        for (int j1 = i1 - d1; j1 <= i1; j1++)
            result += Nu.Basis(j0) * Nv.Basis(j1) * P[j0][j1];
    }
    return result;
}

```

12.5 NURBS RECTANGLE PATCHES

B-spline surface patches are piecewise polynomial functions of two variables. NURBS surface patches are piecewise rational polynomial functions of two variables. Just as for curves, the construction involves fitting homogeneous points in one higher dimension with a B-spline surface ($Y(u, v)$, $w(u, v)$), then projecting back to your application space by dividing by the $w(u, v)$ term: $X(u, v) = Y(u, v)/w(u, v)$. NURBS surfaces have greater flexibility than B-spline surfaces.

A NURBS rectangle surface patch is built from control points P_{i_0, i_1} and weights w_{i_0, i_1} for $0 \leq i_0 \leq n_0$ and $0 \leq i_1 \leq n_1$. The degrees d_i are user selected with $1 \leq d_i \leq n_i$. The surface patch is defined by

$$X(u, v) = \frac{\sum_{i_0=0}^{n_0} \sum_{i_1=0}^{n_1} N_{i_0, d_0}(u) N_{i_1, d_1}(v) w_{i_0, i_1} P_{i_0, i_1}}{\sum_{i_0=0}^{n_0} \sum_{i_1=0}^{n_1} N_{i_0, d_0}(u) N_{i_1, d_1}(v) w_{i_0, i_1}} \quad (12.2)$$

The B-spline construction in one higher dimension uses the homogeneous control points $(w_{i_0, i_1} P_{i_0, i_1}, w_{i_0, i_1})$.

We can encapsulate NURBS rectangle patches into a class `NURBSSurface` and give it two `BasisFunction` members, just like you can do for `BSplineSurface`. The class manages the control points `P[] []` and the control weights `W[] []`. The surface evaluation is

```

Point NURBSSurface::X (float u, float v)
{
    int i0 = Nu.Compute(u), i1 = Nv.Compute(v);
    Point result = ZERO;
    float totalW = 0;
    for (int j0 = i0 - d0; j0 <= i0; j0++)
    {
        for (int j1 = i1 - d1; j1 <= i1; j1++)

```



```

    {
        float tmp = Nu.Basis(j0) * Nv.Basis(j1) * W[j0][j1];
        result += tmp * P[j0][j1];
        totalW += tmp;
    }
}
result /= totalW;
return result;
}

```

12.6 SURFACES BUILT FROM CURVES

In order to avoid the complexity of dealing with a naturally defined surface patch such as a B-spline or a NURBS rectangle patch, sometimes it is convenient to build a surface from curves. The idea is that the curves are easier to work with and potentially lead to less expensive dynamic updates of the surface. A few types of surfaces built from curves are described here. In all cases the parameter space is $(u, v) \in [0, 1]$.

12.6.1 CYLINDER SURFACES

Surface patches might provide more curvature variation than is needed for a particular model. For example, a curved archway is curved in one dimension and flat in another. A single curve may be built to represent the curved dimension, then extruded linearly for the flat dimension. The surface obtained by this operation is said to be a *cylinder surface*. Figure 12.4 illustrates the process.

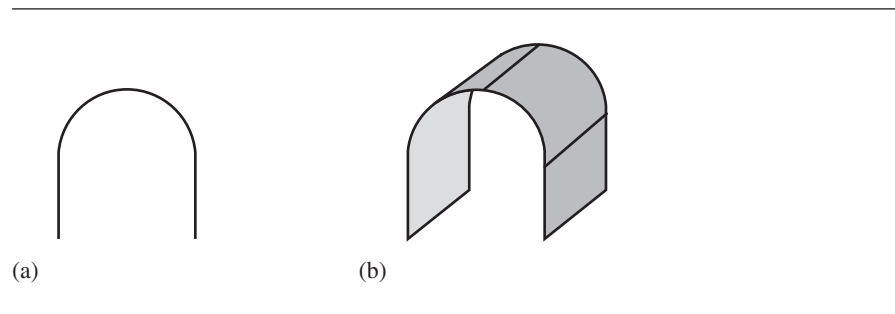


Figure 12.4 A cylinder surface (b) obtained by extruding the curve (a) in a direction oblique to the plane of the curve.

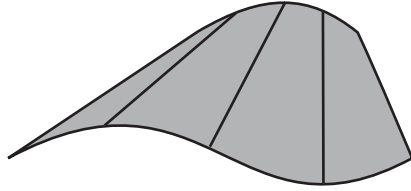


Figure 12.5 A generalized cylinder surface obtained by linearly interpolating pairs of points on two curves.

If $\mathbf{Y}(u)$ is a parameterization of the curve for $u \in [0, 1]$, and if $\mathbf{D}$ is the desired amount of linear translation of the curve, the cylinder surface is parameterized by

$$\mathbf{X}(u, v) = \mathbf{Y}(u) + v\mathbf{D}$$

for $v \in [0, 1]$. First-order partial derivatives are $\partial\mathbf{X}/\partial u = \mathbf{Y}'(u)$ and $\partial\mathbf{X}/\partial v = \mathbf{D}$. Normal vectors to the surface are the cross product of the derivatives,

$$\mathbf{N}(u) = \frac{\mathbf{Y}'(u) \times \mathbf{D}}{|\mathbf{Y}'(u) \times \mathbf{D}|}$$

Notice that the normal does not depend on v .

12.6.2 GENERALIZED CYLINDER SURFACES

Some applications might require that a starting and ending curve be specified and an interpolation applied between them to generate a surface. This is called a *generalized cylinder surface*. Figure 12.5 illustrates.

If $\mathbf{Y}_0(u)$ and $\mathbf{Y}_1(u)$ are the starting and ending curves, $u \in [0, 1]$, the generalized cylinder surface is parameterized by

$$\mathbf{X}(u, v) = (1 - v)\mathbf{Y}_0(u) + v\mathbf{Y}_1(u)$$

for $v \in [0, 1]$. The first-order derivatives are $\partial\mathbf{X}/\partial u = (1 - v)\mathbf{Y}_0'(u) + v\mathbf{Y}_1'(u)$ and $\partial\mathbf{X}/\partial v = \mathbf{Y}_1(u) - \mathbf{Y}_0(u)$. Normal vectors to the surface are

$$\mathbf{N}(u, v) = \frac{((1 - v)\mathbf{Y}_0'(u) + v\mathbf{Y}_1'(u)) \times (\mathbf{Y}_1(u) - \mathbf{Y}_0(u))}{|((1 - v)\mathbf{Y}_0'(u) + v\mathbf{Y}_1'(u)) \times (\mathbf{Y}_1(u) - \mathbf{Y}_0(u))|}$$

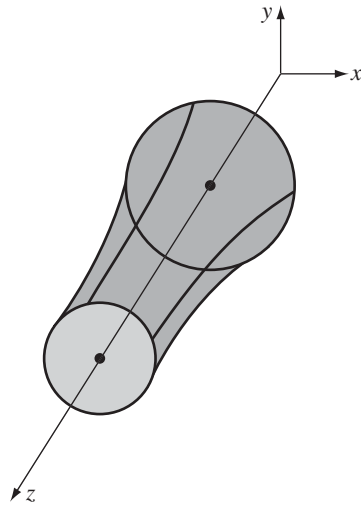


Figure 12.6 A surface of revolution.

12.6.3 REVOLUTION SURFACES

A *revolution surface* is obtained by revolving a curve about a line that does not intersect the curve. To simplify the discussion, suppose that the line is the z -axis and the curve is $(x(u), z(u))$ in the xz -plane. The parameter $u \in [0, 1]$ and $x(u) > 0$. The intersection of the surface and a plane of constant z , given by $z(u)$ for a specified u , is a circle whose radius is $x(u)$, as shown by Figure 12.6.

The surface is parameterized as

$$\mathbf{X}(u, v) = (x(u) \cos(2\pi v), x(u) \sin(2\pi v), z(u))$$

for $(u, v) \in [0, 1]^2$.

12.6.4 TUBE SURFACES

A surface in the shape of a tube can be generated by specifying the central curve of the tube, say, $\mathbf{C}(v)$ for $v \in [0, 1]$, and by specifying a closed planar curve $\mathbf{Y}(u) = (y_1(u), y_2(u))$ to represent the boundary of a cross section of the surface. The cross section for a given v is within a plane whose coordinate system has origin $\mathbf{C}(v)$ and one unit-length coordinate direction $\mathbf{T}(v) = \mathbf{C}'(v)/|\mathbf{C}'(v)|$, a tangent to the central

curve. The other two unit-length coordinate directions are chosen as desired; call them $\mathbf{N}(v)$ and $\mathbf{B}(v)$. The three vectors form a right-handed orthonormal set. The names are suggestive of using the Frenet frame for the curve, where $\mathbf{N}$ is the curve normal and $\mathbf{B} = \mathbf{T} \times \mathbf{N}$ is the curve binormal. However, other choices are always possible. The tube surface is constructed as

$$\mathbf{X}(u, v) = \mathbf{C}(v) + y_1(u)\mathbf{N}(v) + y_2(u)\mathbf{B}(v)$$

for $(u, v) \in [0, 1]^2$. The classical tube surface is one whose cross sections are circular, $\mathbf{Y}(u) = r(\cos u, \sin u)$, for a positive radius r . More generally, the radius can be allowed to vary with v . For example, a surface of revolution is a tube surface whose central curve is a line segment and whose radius varies based on the curve that was revolved about the line segment.

12.7 PARAMETRIC SUBDIVISION

Subdivision is an important process for converting surface patches to a set of triangles that the game engine can use. This section describes subdivision algorithms for rectangle patches, triangle patches, cylinder patches, and spheres or ellipsoids. Two variations of subdivision are considered—uniform and nonuniform subdivision.

12.7.1 SUBDIVISION OF RECTANGLE PATCHES

The ideas of subdivision are best illustrated when the surface patch is a rectangle patch, whether the subdivision is uniform or nonuniform.

Uniform Subdivision

A rectangle patch can be subdivided by uniformly tessellating the parameter space to a specified level $L \geq 0$. Figure 12.7 illustrates the subdivisions for $L = 0$ and $L = 1$. The vertices occur at (s_i, t_j) , where $s_i = i/2^L$ for $0 \leq i \leq 2^L$ and $t_j = j/2^L$ for $0 \leq j \leq 2^L$. The number of vertices in the tessellation is $V = (2^L + 1)^2$, and the number of triangles is $T = 2 \cdot 4^L$.

The obvious way to compute the vertices is iteration of a double loop:

```
L = levels of subdivision;
P = pow(2,L); // maximum index per row or column
M = pow(2,L) + 1; // number of vertices per row or column
vertex[M][M] = array of vertices;
for (i = 0; i < M; i++)
```

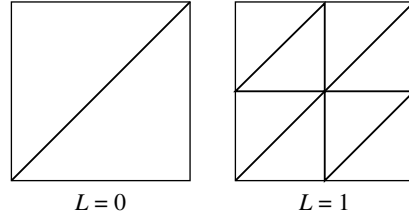


Figure 12.7 Subdivisions of parameter space for a rectangle patch.

```

{
    s = i/P;
    for (j = 0; j < M; j++)
    {
        t = j/P;
        vertex[i][j] = X(s,t); // evaluation of patch
    }
}

```

However, this does not minimize the number of floating-point operations for Bézier rectangle patches of odd degree. Let's consider an example for bicubic patches. Ignoring loop overhead and the divisions for computing s and t (these can be replaced by incrementing by a precomputed delta), the floating-point operations occur in the evaluation of the Bézier patch $\mathbf{X}(s, t)$. Define

$$\begin{aligned}\mathbf{q}_i &= \sum_{j=0}^3 B_{3,j}(t) \mathbf{P}_{i,j} = \mathbf{R}_0^{(i)} + t\mathbf{R}_1^{(i)} + t^2\mathbf{R}_2^{(i)} + t^3\mathbf{R}_3^{(i)} \\ &= \mathbf{R}_0^{(i)} + t(\mathbf{R}_1^{(i)} + t(\mathbf{R}_2^{(i)} + t\mathbf{R}_3^{(i)}))\end{aligned}$$

for $0 \leq i \leq 3$. The $\mathbf{R}_j^{(i)}$ are precomputed. Evaluation of this vector-valued polynomial requires nine multiplications and nine additions. Doing so for each i requires 72 operations. The next evaluation is for

$$\mathbf{X}(s, t) = \sum_{i=0}^3 B_{3,i}(s) \mathbf{Q}_i = \mathbf{C}_0 + s\mathbf{C}_1 + s^2\mathbf{C}_2 + s^3\mathbf{C}_3 = \mathbf{C}_0 + s(\mathbf{C}_1 + s(\mathbf{C}_2 + s\mathbf{C}_3))$$

This requires an additional 18 operations. The total operation count is $90V = 90(2^L + 1)^2$.

A recursive subdivision using central differences may be used just as was done for Bézier cubic curves. Tensor notation is used to simplify the expressions. Rather than explicitly writing summation signs, if an expression contains a repeated index, the assumption is that the index is summed over the appropriate range of values. For example, if $A = [A_{ij}]$ is an $n \times n$ matrix and $\mathbf{X} = [x_j]$ is an $n \times 1$ vector, then the expression $A\mathbf{X}$ is written as $\sum_{j=0}^{n-1} A_{ij}x_j$ in the standard notation, but as $A_{ij}x_j$ using the summation convention. The index j is repeated, so an implied summation occurs over j . The second part of tensor notation specifies derivatives using indices. If $\mathbf{X}(\mathbf{P})$ is an $n \times 1$ vector-valued function of the $m \times 1$ vector $\mathbf{P}$, then the derivative of the i th component of $\mathbf{X}$ with respect to the j th component of $\mathbf{P}$ is denoted $x_{i,j}$. In tensor notation, indices before the subscripted comma refer to components and indices after the comma refer to derivatives. Second derivatives have two indices after the comma, third derivatives have three, and so on.

For a polynomial curve of degree at most three, the identities equivalent to Equation (11.27) for surfaces are

$$\begin{aligned} \mathbf{X}(s, t) &= \frac{1}{2} \left(\mathbf{X}(s + \delta, t) + \mathbf{X}(s - \delta, t) - \delta^2 \mathbf{X}_{ss}(s, t) \right) \\ \mathbf{X}(s, t) &= \frac{1}{2} \left(\mathbf{X}(s, t + \delta) + \mathbf{X}(s, t - \delta) - \delta^2 \mathbf{X}_{tt}(s, t) \right) \end{aligned} \quad (12.3)$$

Similarly, the identities equivalent to Equation (11.28) for surfaces are

$$\begin{aligned} \mathbf{X}_{ss}(s, t) &= \frac{1}{2} \left(\mathbf{X}_{ss}(s + \delta, t) + \mathbf{X}_{ss}(s - \delta, t) \right) \\ \mathbf{X}_{tt}(s, t) &= \frac{1}{2} \left(\mathbf{X}_{tt}(s, t + \delta) + \mathbf{X}_{tt}(s, t - \delta) \right) \end{aligned} \quad (12.4)$$

Now we will describe the algorithm for the block with parameter values $s \in [s_0, s_1]$ and $t \in [t_0, t_1]$. Define $s_m = (s_0 + s_1)/2$, $t_m = (t_0 + t_1)/2$, and $d = s_m - s_0 = t_m - t_0$. At each of the four corner points, it is assumed that the following quantities are precomputed: $\mathbf{X}$, $\mathbf{X}_{ss}$, $\mathbf{X}_{tt}$, and $\mathbf{X}_{sstt}$. The subscripts indicate partial derivatives with respect to the listed variables. The formulas shown below are valid because of Equations (12.3) and (12.4).

For midpoints $(s_m, \bullet)$, where $\bullet$ is either t_0 or t_1 :

$$\begin{aligned} \mathbf{X}_{ss}(s_m, \bullet) &= 0.5 \left(\mathbf{X}_{ss}(s_0, \bullet) + \mathbf{X}_{ss}(s_1, \bullet) \right) \\ \mathbf{X}_{sstt}(s_m, \bullet) &= 0.5 \left(\mathbf{X}_{sstt}(s_0, \bullet) + \mathbf{X}_{sstt}(s_1, \bullet) \right) \\ \mathbf{X}_{tt}(s_m, \bullet) &= 0.5 \left(\mathbf{X}_{tt}(s_0, \bullet) + \mathbf{X}_{tt}(s_1, \bullet) - d^2 \mathbf{X}_{sstt}(s_m, \bullet) \right) \\ \mathbf{X}(s_m, \bullet) &= 0.5 \left(\mathbf{X}(s_0, \bullet) + \mathbf{X}(s_1, \bullet) - d^2 \mathbf{X}_{ss}(s_m, \bullet) \right) \end{aligned}$$

For midpoints $(\bullet, t_m)$, where $\bullet$ is either s_0 or s_1 :

$$\mathbf{X}_{tt}(\bullet, t_m) = 0.5 (\mathbf{X}_{tt}(\bullet, t_0) + \mathbf{X}_{tt}(\bullet, t_1))$$

$$\mathbf{X}_{sstt}(\bullet, t_m) = 0.5 (\mathbf{X}_{sstt}(\bullet, t_0) + \mathbf{X}_{sstt}(\bullet, t_1))$$

$$\mathbf{X}_{ss}(\bullet, t_m) = 0.5 (\mathbf{X}_{ss}(\bullet, t_0) + \mathbf{X}_{ss}(\bullet, t_1) - d^2 \mathbf{X}_{sstt}(\bullet, t_m))$$

$$\mathbf{X}(\bullet, t_m) = 0.5 (\mathbf{X}(\bullet, t_0) + \mathbf{X}(\bullet, t_1) - d^2 \mathbf{X}_{tt}(\bullet, t_m))$$

At the center point (s_m, t_m) :

$$\mathbf{X}_{ss}(s_m, t_m) = 0.5 (\mathbf{X}_{ss}(s_0, t_m) + \mathbf{X}_{ss}(s_1, t_m))$$

$$\mathbf{X}_{tt}(s_m, t_m) = 0.5 (\mathbf{X}_{tt}(s_m, t_0) + \mathbf{X}_{tt}(s_m, t_1))$$

$$\mathbf{X}_{sstt}(s_m, t_m) = 0.5 (\mathbf{X}_{sstt}(s_0, t_m) + \mathbf{X}_{sstt}(s_1, t_m))$$

$$\mathbf{X}(s_m, t_m) = 0.5 (\mathbf{X}(s_0, t_m) + \mathbf{X}(s_1, t_m) - d^2 \mathbf{X}_{ss}(s_m, t_m))$$

If L full subdivisions are performed, then $M_\ell = 2^\ell(2^{\ell-1} + 1)$ new midpoints and $C_\ell = 4^{\ell-1}$ new centers are generated at subdivision ℓ . The total number of midpoints is

$$M = \sum_{\ell=1}^L 2^\ell(2^{\ell-1} + 1) = \frac{2}{3}(4^L - 1) + 2(2^L - 1)$$

and the total number of center points is

$$C = \sum_{\ell=1}^L 4^{\ell-1} = \frac{1}{3}(4^L - 1)$$

For L subdivisions, the total number of vertices is $(2^L + 1)^2$. The four initial corners and the additional midpoints and centers yields

$$\begin{aligned} 4 + \frac{2}{3}(4^L - 1) + 2(2^L - 1) + \frac{1}{3}(4^L - 1) &= 4 + 4^L - 1 + 2(2^L - 1) \\ &= 4^L + 2 \cdot 2^L + 1 = (2^L + 1)^2 \end{aligned}$$

a verification that the counts on the midpoints and centers are correct.

Calculation of d^2 requires one subtraction and one multiplication per level and is not counted in the operation count, because the number is insignificant compared to the number of subdivision vertices. Calculation of $\mathbf{X}$, $\mathbf{X}_{ss}$, $\mathbf{X}_{tt}$, and $\mathbf{X}_{sstt}$ at each

midpoint takes four additions, two subtractions, and six multiplications per vector component. Add these and multiply by 3 (for the three components) to obtain 36 operations per midpoint. The center calculations take four additions, one subtraction, and five multiplications per vector component, times three components, yields 30 operations per center. The total operation count for the full subdivision is

$$36 \left(\frac{2}{3}(4^L - 1) + 2(2^L - 1) \right) + 30 \left(\frac{1}{3}(4^L - 1) \right) = 34(4^L - 1) + 72(2^L - 1)$$

The high-order term in the loop iteration algorithm is $90 \cdot 4^L$. For the recursive subdivision, it is $34 \cdot 4^L$. Therefore, the recursive algorithm is about 2.64 times faster.

The pseudocode for the algorithm follows.

```
void Subdivide (s0, s1, t0, t1, x[2][2], xss[2][2], xtt[2][2], xsstt[2][2])
{
    // Parameter block is [s0,s1]x[t0,t1].
    // x[i][j] = x(si,tj)
    // xss[i][j] = x_{ss}(si,tj)
    // xtt[i][j] = x_{tt}(si,tj)
    // xsstt[i][j] = x_{sstt}(si,tj)

    d = s1 - s0; // = t1 - t0 since blocks are square
    dsqr = d * d;

    xss_m0 = 0.5 * (xss[0][0] + xss[1][0]);
    xss_m1 = 0.5 * (xss[0][1] + xss[1][1]);
    xsstt_m0 = 0.5 * (xsstt[0][0] + xsstt[1][0]);
    xsstt_m1 = 0.5 * (xsstt[0][1] + xsstt[1][1]);
    xtt_m0 = 0.5 * (xtt[0][0] + xtt[1][0] - dsqr * xsstt_m0);
    xtt_m1 = 0.5 * (xtt[0][1] + xtt[1][1] - dsqr * xsstt_m1);
    x_m0 = 0.5 * (x[0][0] + x[1][0] - dsqr * xss_m0);
    x_m1 = 0.5 * (x[0][1] + x[1][1] - dsqr * xss_m1);
    insert x_m0 and x_m1 in subdivision;

    xtt_0m = 0.5 * (xtt[0][0] + xtt[0][1]);
    xtt_1m = 0.5 * (xtt[1][0] + xtt[1][1]);
    xsstt_0m = 0.5 * (xsstt[0][0] + xsstt[0][1]);
    xsstt_1m = 0.5 * (xsstt[1][0] + xsstt[1][1]);
    xss_0m = 0.5 * (xss[0][0] + xss[0][1] - dsqr * xsstt_0m);
    xss_1m = 0.5 * (xss[1][0] + xss[1][1] - dsqr * xsstt_1m);
    x_0m = 0.5 * (x[0][0] + x[0][1] - dsqr * xtt_0m);
    x_1m = 0.5 * (x[1][0] + x[1][1] - dsqr * xtt_1m);
    insert x_0m and x_1m in subdivision;
```



```

xss_mm = 0.5 * (xss_0m + xss_1m);
xtt_mm = 0.5 * (xtt_m0 + xtt_m1);
xsstt_mm = 0.5 * (xsstt_0m + xsstt_1m);
x_mm = 0.5 * (x_0m + x1m-dsqr * xss_mm);
insert x_mm in subdivision;

```

```

sm = 0.5 * (s0 + s1);
tm = 0.5 * (t0 + t1);

```

```

// subblock [s0,sm]x[t0,tm]
y[0][0] = x[0][0];
y[1][0] = x_m0;
y[0][1] = x_0m;
y[1][1] = x_mm;
yss[0][0] = xss[0][0];
yss[1][0] = xss_m0;
yss[0][1] = xss_0m;
yss[1][1] = xss_mm;
ytt[0][0] = xtt[0][0];
ytt[1][0] = xtt_m0;
ytt[0][1] = xtt_0m;
ytt[1][1] = xtt_mm;
yysstt[0][0] = xsstt[0][0];
yysstt[1][0] = xsstt_m0;
yysstt[0][1] = xsstt_0m;
yysstt[1][1] = xsstt_mm;
Subdivide(s0,sm,t0,tm,y,yss,ytt,yysstt);

```

```

// subblock [s0,sm]x[tm,t1]
y[0][0] = x_0m;
y[1][0] = x_mm;
y[0][1] = x[0][1];
y[1][1] = x_m1;
yss[0][0] = xss_0m;
yss[1][0] = xss_mm;
yss[0][1] = xss[0][1];
yss[1][1] = xss_m1;
ytt[0][0] = xtt_0m;
ytt[1][0] = xtt_mm;
ytt[0][1] = xtt[0][1];
ytt[1][1] = xtt_m1;

```

```

ysstt[0][0] = xsstt_0m;
ysstt[1][0] = xsstt_mm;
ysstt[0][1] = xsstt[0][1];
ysstt[1][1] = xsstt_m1;
Subdivide(s0,sm,tm,t1,y,yss,ytt,ysstt);

// subblock [sm,s1]x[t0,tm]
y[0][0] = x_m0;
y[1][0] = x[1][0];
y[0][1] = x_mm;
y[1][1] = x_lm;
yss[0][0] = xss_m0;
yss[1][0] = xss[1][0];
yss[0][1] = xss_mm;
yss[1][1] = xss_lm;
ytt[0][0] = xtt_m0;
ytt[1][0] = xtt[1][0];
ytt[0][1] = xtt_mm;
ytt[1][1] = xtt_lm;
ysstt[0][0] = xsstt_m0;
ysstt[1][0] = xsstt[1][0];
ysstt[0][1] = xsstt_mm;
ysstt[1][1] = xsstt_lm;
Subdivide(sm,s1,t0,tm,y,yss,ytt,ysstt);

// subblock [sm,s1]x[tm,t1]
y[0][0] = x_mm;
y[1][0] = x_lm;
y[0][1] = x_m1;
y[1][1] = x[1][1];
yss[0][0] = xss_mm;
yss[1][0] = xss_lm;
yss[0][1] = xss_m1;
yss[1][1] = xss[1][1];
ytt[0][0] = xtt_mm;
ytt[1][0] = xtt_lm;
ytt[0][1] = xtt_m1;
ytt[1][1] = xtt[1][1];
ysstt[0][0] = xsstt_mm;
ysstt[1][0] = xsstt_lm;
ysstt[0][1] = xsstt_m1;
ysstt[1][1] = xsstt[1][1];
Subdivide(sm,s1,tm,t1,y,yss,ytt,ysstt);
}

```

Warning: The code does not show how to memoize the various quantities so that terms are not computed multiple times. If the initial block is subdivided into four subblocks, the code as shown will twice compute the quantities at the midpoint of the shared edge $\{s_m\} \times [t_0, t_m]$, once for subblock $[s_0, s_m] \times [t_0, t_m]$ and once for subblock $[s_m, s_1] \times [t_0, t_m]$. One possibility for avoiding the repetitive calculations is to assign responsibility for the various midpoint quantities to specific subblocks and to pass an additional parameter to `Subdivide` that indicates which of the four subblocks is being recursed on. Block $[s_0, s_m] \times [t_0, t_m]$ is responsible for four midpoints, blocks $[s_0, s_m] \times [t_m, t_1]$ and $[s_m, s_1] \times [t_0, t_m]$ are each responsible for three midpoints, and block $[s_m, s_1] \times [t_m, t_1]$ is responsible for two midpoints.

Nonuniform Subdivision

The recursive uniform subdivision ignores two important aspects of rendering surfaces. The first aspect is that the patch may be relatively flat in some subblocks. There is no point in further subdividing those subblocks, because no additional variation is to be found in the surface. The second aspect is that the surface might be far away from the eye point. A fixed level of subdivision could produce a suitable number of triangles to accurately represent the surface when near the eye point, but the same level might produce a large number of small triangles that are expensive to render yet do not contribute much to the perceived shape of the patch. A smarter subdivision scheme will handle both aspects appropriately.

The recursive subdivision can be modified to terminate at a block if the measured variation within the block is insignificant. This modification is done much in the same way as the recursive algorithm for curves. If any of the second derivatives at the four midpoints of a block's edges is significantly large, then the block is subdivided. If the second derivatives at the four midpoints are all significantly small, then the block is not subdivided. For recursive uniform subdivision, the algorithm essentially builds a complete quadtree of the specified level. The modified recursive algorithm builds a partial quadtree.

The term in Equation (12.3) that measures variation from the line segment connecting the endpoints of the interval is $x_{i,jk}(\mathbf{P})\delta_j\delta_k$, with the factor of one-half omitted. According to the summation convention, there is a double summation over indices j and k . The remaining index i is a free index, so this quantity is a vector; call it $\mathbf{V}$. For a midpoint calculation on a horizontal edge $[s_0, s_1]$, let $\Delta = s_1 - s_0$; then $\delta = \Delta(1, 0)$ and $\mathbf{V} = \Delta^2 \mathbf{X}_{ss}((s_0 + s_1)/2, t)$. For a midpoint calculation on a vertical edge $[t_0, t_1]$, let $\Delta = t_1 - t_0$; then $\delta = \Delta(0, 1)$ and $\mathbf{V} = \Delta^2 \mathbf{X}_{tt}(s, (t_0 + t_1)/2)$.

The pseudocode for the unconstrained recursion can be modified to add the tests on the size of $\mathbf{V}$.

```
void Subdivide (s0, s1, t0, t1, x[2][2], xss[2][2], xtt[2][2], xsstt[2][2])
{
    // Parameter block is [s0,s1]x[t0,t1].
```

```

// x[i][j] = x(si,tj)
// xss[i][j] = x_{ss}(si,tj)
// xtt[i][j] = x_{tt}(si,tj)
// xsstt[i][j] = x_{sstt}(si,tj)

d = s1 - s0; // = t1 - t0 since blocks are square
dsqr = d * d;

xss_m0 = 0.5 * (xss[0][0] + xss[1][0]);
xss_m1 = 0.5 * (xss[0][1] + xss[1][1]);
xtt_0m = 0.5 * (xtt[0][0] + xtt[0][1]);
xtt_1m = 0.5 * (xtt[1][0] + xtt[1][1]);

vm0 = dsqr * xss_m0;
vm1 = dsqr * xss_m1;
v0m = dsqr * xtt_0m;
v1m = dsqr * xtt_1m;

if ( SquaredLength(vm0) > epsilon or
    SquaredLength(vm1) > epsilon or
    SquaredLength(v0m) > epsilon or
    SquaredLength(v1m) > epsilon )
{
    // Subdivide the block.

    xsstt_m0 = 0.5 * (xsstt[0][0] + xsstt[1][0]);
    xsstt_m1 = 0.5 * (xsstt[0][1] + xsstt[1][1]);
    xtt_m0 = 0.5 * (xtt[0][0] + xtt[1][0] - dsqr * xsstt_m0);
    xtt_m1 = 0.5 * (xtt[0][1] + xtt[1][1] - dsqr * xsstt_m1);
    x_m0 = 0.5 * (x[0][0] + x[1][0] - dsqr * xss_m0);
    x_m1 = 0.5 * (x[0][1] + x[1][1] - dsqr * xss_m1);
    insert x_m0 and x_m1 in subdivision;

    xsstt_0m = 0.5 * (xsstt[0][0] + xsstt[0][1]);
    xsstt_1m = 0.5 * (xsstt[1][0] + xsstt[1][1]);
    xss_0m = 0.5 * (xss[0][0] + xss[0][1] - dsqr * xsstt_0m);
    xss_1m = 0.5 * (xss[1][0] + xss[1][1] - dsqr * xsstt_1m);
    x_0m = 0.5 * (x[0][0] + x[0][1] - dsqr * xtt_0m);
    x_1m = 0.5 * (x[1][0] + x[1][1] - dsqr * xtt_1m);
    insert x_0m and x_1m in subdivision;

    xss_mm = 0.5 * (xss_0m + xss_1m);
    xtt_mm = 0.5 * (xtt_m0 + xtt_m1);
    xsstt_mm = 0.5 * (xsstt_0m + xsstt_1m);

```

```

    x_mm = 0.5 * (x_0m + x1m-dsqr * xss_mm);
    insert x_mm in subdivision;

    sm = 0.5 * (s0 + s1);
    tm = 0.5 * (t0 + t1);

    // The pseudocode from the unconstrained algorithm for the
    // four subblocks goes here....
  }
}

```

This pseudocode has the same warning as for the unconstrained case. The various midpoint quantities could be computed twice. Note that assigning responsibility for computing the various midpoint quantities to specific subblocks does not work in this case. The problem is that one subblock decides not to recurse on its children, thereby not calculating some of the midpoint quantities (the ones that occur in the “subdivide the block” chunk of code), but a neighboring subblock relies on these values being computed. A modification that takes care of this is to provide a set of Boolean flags indicating which of the midpoint quantities still requires computation. By using these, we effectively have a classic table of memoized values. Another possibility is to allow the multiple computations to occur. The worst case is that all midpoints are calculated twice. The number of midpoints to compute at level ℓ is $M_\ell = 4^\ell$. The total number of midpoints for L levels of subdivision is $M = \sum_{\ell=1}^L M_\ell = 4(4^L - 1)/3$. The total number of center points is $C = (4^L - 1)/3$. The total operation count (see the formula for the unconstrained case) is $36M + 30C = 58(4^L - 1)$. The high-order term for the loop iteration algorithm was $90 \cdot 4^L$, for the unconstrained algorithm was $34 \cdot 4^L$, and for the current algorithm is $58 \cdot 4^L$. The approximate speedup over the loop iteration is 1.55—still faster, but not as fast as the algorithm that avoids the repetitious calculations.

Adjustments for the Camera Model

The nonuniform subdivision tests the lengths of the nonlinear terms $\mathbf{V}$ to decide whether or not to subdivide. For a surface with a lot of variation in it, the subdivisions will occur. In the presence of a camera model and perspective projection, the subdivision is acceptable when the surface is near the eye point. However, if the surface is far away from the eye point, the subdivision may not add much to the visual quality of the rendered surface because each already existent triangle maps only to a handful of pixels on the screen.

One heuristic for the subdivision step is provided in [Sha99]. The idea is to get an estimate of the length (in pixels) of the projection of $\mathbf{V}$ into screen space. [Sha99] estimates a slice, perpendicular to the camera direction, in the view frustum in which $\mathbf{V}$ lives, computes the width (in pixels) of that slice, then computes the ratio of the

length of $\mathbf{V}$ to the slice width and compares that ratio to a tolerance. If smaller, the subdivision step is performed.

Another possibility is to compute the midpoint $\mathbf{M}$ of the line segment connecting the two known endpoints and compute the length of the projected line segment (in pixels) from $\mathbf{M}$ to $\mathbf{M} - \mathbf{V}/2$, the last point being the actual midpoint if the edge were to be subdivided. If that distance is larger than an application-specified number of pixels, then the subdivision is performed.

Cracking

The story is not yet finished. Nonuniform subdivision allows for neighboring blocks to be subdivided to different resolutions, which creates cracking in the final mesh. Figure 12.8 shows two adjacent blocks in a subdivision that has cracking. The crack occurs at the T-junction marked with a solid dot. The left block wants to be subdivided, but the right block does not. It appears as if one of two choices can be made, and the consequences of either are undesirable. The first choice is to subdivide those blocks that want to be subdivided and force adjacent blocks to follow suit. Applied recursively in the quadtree, this will force a uniform tessellation to the level of the most detailed block. The second choice is to disallow subdivision of blocks that have an adjacent block that does not want to subdivide. This will also force a uniform tessellation, but to the level of the least detailed block. The problem here is subdivision performed strictly as a quadtree process. To accommodate adjacent blocks that do not jointly subdivide (in the quadtree sense), we need to allow for a form of *partial subdivision*. Even with a suitable definition for partial subdivision, the same two choices remain about whether to force the least detailed block to partially subdivide or to prevent the most detailed block from partially subdividing.

An approach that prevents the quadtree subdivision of a more detailed block is mentioned in [Sha99]. That article illustrates how to resolve the cracking shown in Figure 12.8. The idea is to collapse the midpoint vertex to a corner vertex, as shown in Figure 12.9. The general algorithm can be stated as follows. For each block, if all of the edges are at the same level of detail, then no collapsing is required. Otherwise, collapse

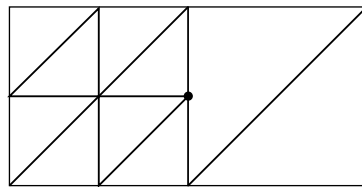


Figure 12.8 Subdivision that contains cracking.

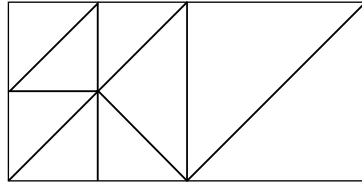


Figure 12.9 Subdivision that has no cracking.

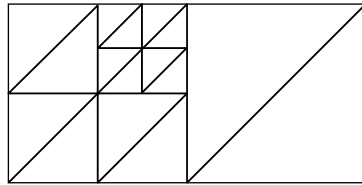


Figure 12.10 Subdivision that contains more complicated cracking.

the midpoints to the corners at those edges. Recurse on the four subblocks. The nonuniform subdivision requires two passes: one to generate the vertices of the final mesh and one to fix the cracking. While the second pass may not be that expensive, it must handle the type of subdivision shown in Figure 12.10.

Now let us consider a single-pass algorithm that prevents the quadtree subdivision of a more detailed block. The main idea is to do a depth-first traversal of the quadtree, but to use topological information about neighboring blocks to decide if the traversal can continue at the current block. Since neighboring blocks might not have been visited yet, the topological information is obtained by allowing a block to compute quantities that the neighbor would have computed if it had been visited first in the traversal. To avoid recalculating that information, a temporary buffer is used to store computed vertices. The buffer is shared by all patches in the system, so the per-object memory costs are avoided. Consider first the block corresponding to the root of the quadtree. If all four edges want to subdivide, then the block is subdivided into four subblocks and the subdivision process is applied to those subblocks.

Suppose that the right edge of the root block does not want to be subdivided. The partial subdivision is illustrated in Figure 12.11. The upper-right and lower-right subblocks are no longer considered for subdivision as the tessellation of that part of the parameter space is already determined to be the three triangles that are shown in the figure. The partial subdivision allows the upper-left and lower-left subblocks

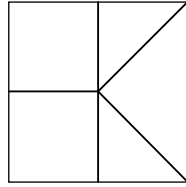


Figure 12.11 Partial subdivision with three subdividing edges.

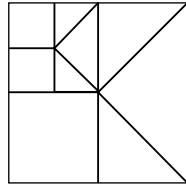


Figure 12.12 Partial subdivision illustrating the parent's topological constraint.

to continue subdividing, but with constraints. The right edges of those subblocks cannot subdivide because of the final tessellation in the two neighboring subblocks. At best, the topology of the partial subdivision for either subblock can look only like that of the parent block. Figure 12.12 illustrates this for the upper-left subblock. The topological constraints for the subdivision of the child blocks are actually quite natural. The right half of the original block is relatively flat since the right edge did not want to subdivide. The left half of the original block is less flat and wants to subdivide to show off its detail. The constraints lead to a tessellation that conforms to the demands of both halves with a gradual increase in tessellation from right to left.

Suppose that both the right and bottom edges of the root block do not want to be subdivided. The partial subdivision is illustrated in Figure 12.13. The upper-right, lower-right, and lower-left subblocks are no longer considered for subdivision. The tessellation for that part of the parameter space is determined to be the four triangles shown in the figure. The partial subdivision allows the upper-left subblock to continue subdividing, but again with constraints. The right and bottom edges cannot be subdivided, just as the parent's edges cannot be subdivided. At best, the topology of the partial subdivision can look only like that of the parent block. Figure 12.14 illustrates this for the subblock. The constraint allows a gradual increase in detail from lower right to upper left.

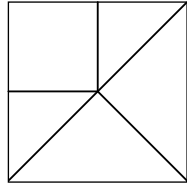


Figure 12.13 Partial subdivision with two adjacent subdividing edges.

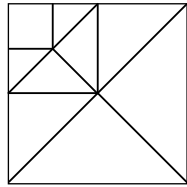


Figure 12.14 Partial subdivision illustrating the parent's topological constraint.

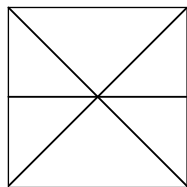


Figure 12.15 Partial subdivision with two opposing subdividing edges.

Suppose that both the top and bottom edges of the root block do not want to be subdivided. None of the subblocks are considered for subdivision. However, the left and right triangles in the partial subdivision can be split in half. Figure 12.15 illustrates the partial subdivision with the two additional triangle splits. The surface appears to have saddlelike behavior in the block. No further subdivision is necessary to explore this feature.

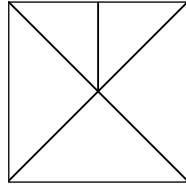


Figure 12.16 Partial subdivision with one subdividing edge.

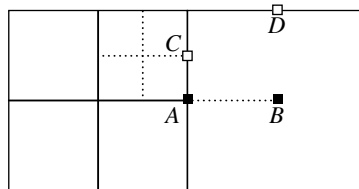


Figure 12.17 Subdivision based on calculating information in an adjacent block.

Finally, suppose that only the top edge wants to be subdivided. None of the subblocks are considered for subdivision and the top triangle can be split in half. Figure 12.16 illustrates the subdivision with the additional triangle split.

Figure 12.17 illustrates how a block can subdivide by calculating information in an adjacent block. The left block L is visited first in the quadtree traversal. If L determines that its right edge can be subdivided, then point A is computed. Block R shares that edge and would have agreed to split and compute A also. Because A occurs in the subdivision, point B must occur in the subdivision of R , so L computes it for R . The children of block L are traversed next. The upper-right child might want to subdivide its right edge and compute point C , but this split is allowed by block R only if point D occurs in the subdivision of R . Since R has not yet been visited in the traversal, L can go ahead and determine if the top edge of R can be split. If so, D is computed and the recursion on the upper-right child of L is allowed. A shared array of subdivision points is used by all patches for temporary storage, and a shared array of Boolean flags is used to indicate whether or not a subdivision point has been computed. Before subdivision of a patch, the Boolean array has all its entries set to false.

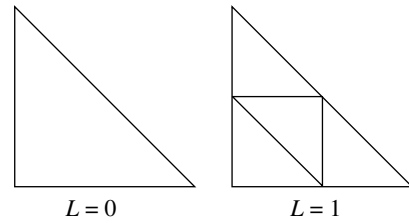


Figure 12.18 Subdivisions of parameter space for a triangle patch.

12.7.2 SUBDIVISION OF TRIANGLE PATCHES

Triangle patches are more difficult to subdivide than rectangle patches because of the more complicated indexing. However, the concepts are still the same at a high level.

Uniform Subdivision

A triangle patch can be subdivided by uniformly tessellating the parameter space to a specified level $L \geq 0$. Figure 12.18 illustrates the subdivisions for $L = 0$ and $L = 1$. For general level $L > 0$, the number of vertices in the tessellation is $V = (2^L + 1)(2^{L-1} + 1)$, and the number of triangles is $T = 4^L$. The vertices are packed in a 1D array, the bottom row first (containing $2^L + 1$ vertices) through the top row last (containing 1 vertex). The mapping from the integer lattice point (x, y) , where x and y are nonnegative integers with $x + y \leq 2^L$, to a 1D array index i is

$$i = x + \frac{y(2^{L+1} + 3 - y)}{2}$$

The straightforward way to compute the vertices is iteration of a double loop. The vertices are stored in a 1D array as mentioned previously. The following pseudocode also shows how to generate an array of indices that represents the triangle connectivity. Each group of three indices corresponds to those vertices that make up a triangle in the tessellation.

```
N = pow(2,L);

// Compute the vertices.
k = 0;
for (y = 0; y <= N; y++)
{
    v = y/N;
```

```

for (x = 0; x + y <= N; x++)
{
    u = x/N;
    vertex[k++] = X(u,v); // evaluation of triangle patch
}

// Compute the triangle connectivity.
t = 0;
ystart = 0;
for (y = 0; y < N; y++)
{
    k0 = ystart;
    k1 = k0 + 1;
    ystart = (y + 1) * (2 * (N + 1) - y)/2;
    k2 = ystart;
    for (x = 0; x + y < N; x++)
    {
        connectivity[t++] = k0;
        connectivity[t++] = k1;
        connectivity[t++] = k2;

        if ( x + y + 1 < N )
        {
            connectivity[t++] = k1;
            connectivity[t++] = k2 + 1;
            connectivity[t++] = k2;
        }

        k0++;
        k1++;
        k2++;
    }
}

```

Nonuniform Subdivision

Like the algorithm of Lindstrom et al. (1996), the following algorithm is based on the ideas in [LKR⁺96]; it uses the equivalent of a symmetric triangulation for quadtree blocks and has a vertex dependency structure.

Consider a single triangle whose vertices are labeled as top T , left L , and right R . If the angle at T is a right angle, the edge from L to R is called the *hypotenuse* of the triangle. Taking liberty with the terminology, even if the angle at T is not a right

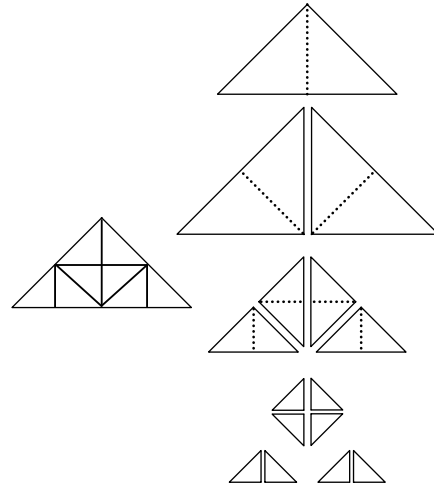
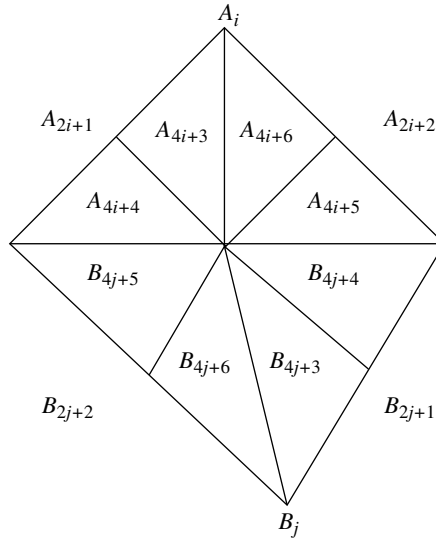


Figure 12.19 Subdivision of a triangle and the corresponding binary tree.

angle, the edge opposite that vertex will be called the hypotenuse. The subdivision algorithm involves deciding if the hypotenuse of a triangle can be subdivided based on the same heuristic as is used for rectangle patch subdivision. If so, the triangle is split into two triangles. The midpoint of the original hypotenuse becomes the top vertex for the two new triangles. The left and right vertices are labeled so that T , L , and R occur in counterclockwise order. The subdivision process is applied to each of the two new subtriangles. For an unconstrained subdivision of a single triangle, the result is a complete binary tree whose leaf nodes represent the final triangles in the subdivision. Figure 12.19 illustrates the subdivision step applied three times.

The labeling of the subdivided triangles is important in the remainder of this section. The triangle for the root node of the tree is labeled A_0 . If A_i is the current triangle to be subdivided and has top vertex T , left vertex L , and right vertex R , and if M is the midpoint of the hypotenuse, then the two children of A_i are A_{2i+1} and A_{2i+2} . The top, left, and right vertices of A_{2i+1} are M , T , and L , respectively. The top, left, and right vertices of A_{2i+2} are M , R , and T . The level in the tree at which A_i occurs is $\ell = \lfloor \log_2(i + 1) \rfloor$, where $\lfloor x \rfloor$ is the floor function that computes the largest integer smaller or equal to x .

The subdivision is more complicated for a triangle mesh. Two adjacent triangles are said to be *H-adjacent* if they share the same hypotenuse. If one of the triangles wants to subdivide its hypotenuse, the other one must also. For a single triangle, this leads to a vertex dependency structure based on the following relationships:

Figure 12.20 H-adjacency for triangles A and B .

- If A_i and A_{i+1} are siblings, then A_{2i+1} and A_{2i+4} are H-adjacent.
- If A_i and A_j are H-adjacent, then A_{4i+4} and A_{4j+5} are H-adjacent. By symmetry, A_{4i+5} and A_{4j+4} are also H-adjacent.

For two triangles A_0 and B_0 that are H-adjacent, the last relationship is also valid. If A_i and B_j are H-adjacent, then A_{4i+4} and B_{4j+5} are H-adjacent and A_{4i+5} and B_{4j+4} are H-adjacent. Figure 12.20 illustrates the relationships.

The relationships between two triangles that are adjacent on an edge that is not the hypotenuse affects the indexing in the H-adjacency. If A_0 has vertices T , L_0 , and R_0 , and if C_0 is a triangle that has top vertex T , left vertex L_1 , and right vertex L_0 , then A_0 and C_0 are not H-adjacent. If M_0 is the midpoint of the edge from L_1 to L_0 , then C_2 has top, left, and right vertices M_0 , L_0 , and T , respectively. If M_1 is the midpoint of the edge from L_0 to R_0 , then A_1 has top, left, and right vertices M_1 , T , and L_0 , respectively. It is the case that A_1 and C_2 are H-adjacent. Generally, if A_{2i+1} and C_{2j+2} are H-adjacent, then so are A_{8i+9} and C_{8j+12} and A_{8i+8} and C_{8j+13} . Figure 12.21 illustrates the relationships. The same constructions apply for an adjacent triangle D_0 whose top, left, and right vertices are T , R_0 , and R_1 , respectively. The triangles A_2 and D_1 are H-adjacent. Generally, if A_{2i+2} and D_{2j+1} are H-adjacent, then so are A_{8i+13} and D_{8j+8} and A_{8i+12} and D_{8j+9} . Figure 12.22 illustrates the relationships.

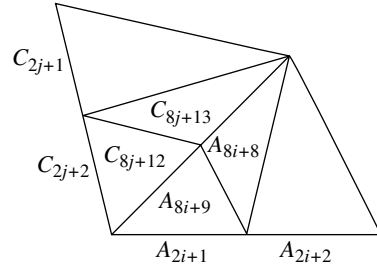


Figure 12.21 H-adjacency for triangles A and C.

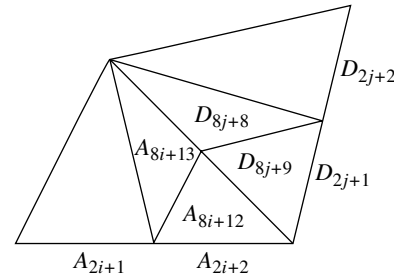


Figure 12.22 H-adjacency for triangles A and D.

For a single triangle to be subdivided, given a maximum level L for subdivision, the storage requirements for vertices are easily computed. Figure 12.23 illustrates the pattern for subdivision for levels $0 \leq L \leq 4$. The number of vertices for maximum subdivision at level L is

$$V = \begin{cases} \sum_{k=1}^{2^{L/2}+1} k, & L \text{ even} \\ \sum_{k=1}^{2^{(L-1)/2}+1} (2k-1), & L \text{ odd} \end{cases} = \begin{cases} (2^{L/2}+1)(2^{L/2-1}+1), & L \text{ even} \\ (2^{(L-1)/2}+1)^2, & L \text{ odd} \end{cases}$$

The number of triangles for maximum subdivision at level L is $T = 2^L$. Vertex storage is as a regular triangular array with row-major indexing, hypotenuse row first through top vertex last. Indexing will depend on the parity of L .

The storage for the worst case can be allocated for a single triangle. Each subdivided triangle uses the same storage and, once subdivided, the renderer must draw the triangles to free the storage for the next triangle to use. This approach leads to some redundant calculations of vertices, those that lie on shared edges of the original

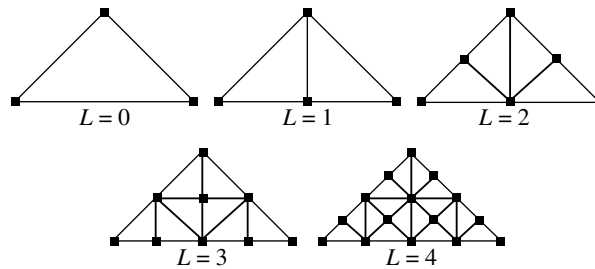
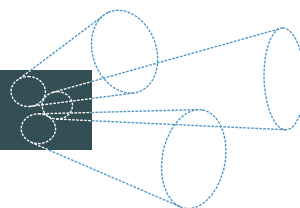


Figure 12.23 Pattern for subdivision of a triangle.

mesh triangles. If size of storage is not an issue, then vertex storage can be allocated per triangle, and the vertex dependencies for H-adjacent triangles can be used in support for calculating each subdivision vertex exactly once. If size of storage is an issue, but speed of vertex calculations is not, then each leaf triangle in the binary tree can be drawn when visited, with effectively no storage requirements. The binary tree is virtually traversed simply by the recursive function calls:

```
void Subdivide (Point T, Point L, Point R)
{
    M = R - L;
    compute second-derivative vector D;
    if ( D is sufficiently large )
    {
        Subdivide(M,T,L);
        Subdivide(M,R,T);
    }
    else
    {
        DrawTriangle(T,L,R);
    }
}
```

Finally, given a triangle mesh, it is necessary to select those edges that will be hypotenuses. It appears that this should always be possible with the mesh, but a guaranteed way of doing this that does not require a preprocessing pass is to subdivide each triangle into three triangles by adding the centroid and edges connecting the centroid to the original vertices. In this way all the edges of the original mesh are the hypotenuses of the tripled mesh. Moreover, if a rectangle mesh is processed in the same way by adding the centroids and connecting to the four corners, the resulting triangle mesh can be subdivided with the algorithm mentioned in this section. This subdivision is the symmetric triangulation discussed in [LKR⁺96] for terrains.



CONTAINMENT METHODS

The following three queries are common in a 3D computer graphics application, mainly in working with bounding volumes for objects, whether for culling or for collision detection.

- Determine if a point is contained in a bounding volume.
- Compute a bounding volume of a set of points.
- Compute a bounding volume that contains a set of bounding volumes (merge bounding volumes).

This section presents some algorithms for each type of bounding volume of interest.

13.1 SPHERES

The queries are organized with the point-in-sphere query first, the computation of bounding volumes second, and the merging of bounding volumes third.

13.1.1 POINT IN SPHERE

The point-in-sphere query is simple enough. If the sphere has center $\mathbf{C}$ and radius r , and if the point is $\mathbf{P}$, then the point is inside the sphere (boundary included) when

$$|\mathbf{P} - \mathbf{C}| \leq r$$

The squared length and the squared radius may be compared instead if you want to avoid the somewhat expensive square root operation that occurs in a length computation.

13.1.2 SPHERE CONTAINING POINTS

Three algorithms are presented for computing a bounding sphere for a set of points.

Sphere Containing Axis-Aligned Box of Points

A simple approach is to compute the minimum-volume, axis-aligned bounding box of the points, then select the smallest enclosing sphere of the box with the sphere centered at the box center. The algorithm is shown next, where points $P[i]$ are indexed by 0 (the x -component), 1 (the y -component), or 2 (the z -component).

```
Vector3 min = P[0], max = min;
for (i = 1; i < n; i++)
{
    for (j = 0; j < 3; j++)
    {
        if (P[i][j] < min[j] )
        {
            min[j] = P[i][j];
        }
        else if (P[i][j] > max[j])
        {
            max[j] = P[i][j];
        }
    }
}
Sphere3 sphere;
sphere.center = (min + max)/2;
Vector3 diagonal = (max - min)/2;
sphere.radius = diagonal.Length();
```

An advantage of this algorithm is the speed with which it is executed. The drawback to this algorithm is that the bounding sphere is not as good a fit as it could be.

Sphere Centered at Average of Points

An alternative that takes longer to compute but provides a somewhat better fit is to select the sphere center to be the average of the points and the sphere radius to be the

smallest value for which the sphere of the given center and that radius encloses the points. The algorithm is

```

Vector3 sum = P[0];
for (i = 1; i < n; i++)
{
    sum += P[i];
}

Sphere3 sphere;
sphere.center = sum/n;

float radiusSqr = 0;
for (i = 0; i <= n; i++)
{
    Vector3 diff = P[i] - sphere.center;
    float temp = diff.SquaredLength();
    if (temp > radiusSqr)
    {
        radiusSqr = temp;
    }
}

sphere.radius = sqrt(radiusSqr);

```

Minimum-Volume Sphere

Computing the minimum-volume sphere that encloses the points requires a more complicated algorithm based on work by [Wel91]. The algorithm uses a randomized linear algorithm, so the execution time is *expected* to be linear in the number of input points. The worst case is polynomial in the number of inputs, but the input data is randomly permuted so that the probability of the worst case occurring is negligible.

The pseudocode for the algorithm given next computes the minimum-volume sphere containing n points $P[0]$ through $P[n - 1]$. The idea is to maintain a set of supporting points for the sphere while processing the input-point set one point at a time. The supporting points lie on the sphere and no other points are necessary to form the sphere. A sphere is supported by two points, by three points, or by four points. Such a sphere is itself the minimum-volume sphere containing the supporting points.

```

Sphere3 ComputeMinimumSphere (int n, Vector3 P[])
{

```

```

// Local storage for the points. Use pointers if your
// programming language supports them.
array<Vector3> Q = P;

// Randomly permute the points. The function
// randomInteger(m) is a hypothetical function that
// generates a random integer between 0 and m. Use
// your own favorite random number generator.
for (i = n - 1; i > 0; i--)
{
    j = randomInteger(i);
    if (j != i) { swap(Q[i],Q[j]); }
}

Sphere3 minSphere = ExactSphere1(Q[0]);
set<Vector3> support = { Q[0] };
i = 1;
while (i < n)
{
    if (Q[i] not in support)
    {
        if (Q[i] not in minSphere)
        {
            // Update the support set and return the
            // bounding sphere for it.
            minimal = Update(i,Q,support);

            // Need to restart the algorithm when the
            // support changes.
            i = 0;
            continue;
        }
    }
    i++;
}
}

```

The function `ExactSphere1` produces a sphere whose center is the input point and whose radius is 0. The function `Update` is the heart of the algorithm. If the support set has k points, where $2 \leq k \leq 4$, and $Q[i]$ is outside the bounding sphere of the support set, then `Update` has the responsibility to look at all possible combinations of $Q[i]$ with the current points in the support set and select the combination that produces the minimum-volume bounding sphere.

Part of the update involves computing the minimum-volume spheres that contain exactly two points, exactly three points, or exactly four points. The term *contain*

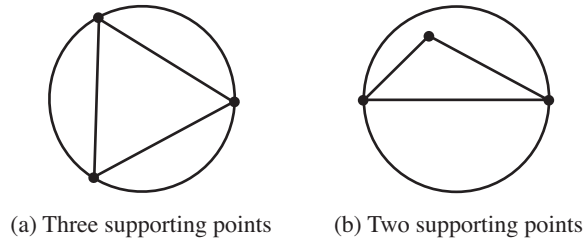


Figure 13.1 (a) The minimum-area circle containing three points and all points are on the circle itself. (b) The minimum-area circle containing three points but only two points are on the circle.

means that the points must be on the sphere boundary, not the interior. The function `ExactSphere2(S0, S1)` computes the minimum-volume sphere containing the inputs. Necessarily, S_0 and S_1 are the endpoints of a diameter of the sphere, so the sphere center is

$$C = (S_0 + S_1)/2$$

and the sphere radius is

$$r = |S_1 - S_0|/2$$

The reason I called the function `ExactSphere2` and not `MinimumSphere2` will become apparent in the discussion of how to handle a set of three points. If you were to implement a function `MinimumSphere3(S0, S1, S2)` to compute the minimum-volume sphere that contains the input points, you would have to deal with degeneracies. For example, if the input points are all the same point, then the output should be the same as for `ExactSphere1`. Unless you have preprocessed your original points to remove duplicates, you must guard against the case when the input points to `MinimumSphere3` are the same. Also, if the three input points are collinear, then only two of them affect the bounding sphere; these are the endpoints of the line segment containing all three points. In this case, you would want the output to be the same as that for `ExactSphere2`.

Now suppose the three inputs are noncollinear. The minimum-volume sphere does not necessarily have all three points on its boundary! Figure 13.1 illustrates this. The points are coplanar, so the circles of intersection of the spheres and the plane are shown.

The circle of Figure 13.1 (a) is generated by simple algebraic means. Let the points be S_0 , S_1 , and S_2 . The center of the circle containing these points has a center written in barycentric coordinates as

$$C = b_0 S_0 + b_1 S_1 + b_2 S_2$$

where $b_i \in [0, 1]$ and $b_0 + b_1 + b_2 = 1$. The center is equidistant from the three points, so

$$|\mathbf{C} - \mathbf{S}_0| = |\mathbf{C} - \mathbf{S}_1| = |\mathbf{C} - \mathbf{S}_2| = r^2$$

where r is the (as of yet unknown) radius of the circle. From these conditions, we have

$$\mathbf{C} - \mathbf{S}_0 = b_0 \mathbf{E}_0 + b_1 \mathbf{E}_1 - \mathbf{E}_0$$

$$\mathbf{C} - \mathbf{S}_1 = b_0 \mathbf{E}_0 + b_1 \mathbf{E}_1 - \mathbf{E}_1$$

$$\mathbf{C} - \mathbf{S}_2 = b_0 \mathbf{E}_0 + b_1 \mathbf{E}_1$$

where $\mathbf{E}_0 = \mathbf{S}_0 - \mathbf{S}_2$ and $\mathbf{E}_1 = \mathbf{S}_1 - \mathbf{S}_2$. These lead to

$$r^2 = |b_0 \mathbf{E}_0 + b_1 \mathbf{E}_1|^2 - 2\mathbf{E}_0 \cdot (b_0 \mathbf{E}_0 + b_1 \mathbf{E}_1) + |\mathbf{E}_0|^2$$

$$r^2 = |b_0 \mathbf{E}_0 + b_1 \mathbf{E}_1|^2 - 2\mathbf{E}_1 \cdot (b_0 \mathbf{E}_0 + b_1 \mathbf{E}_1) + |\mathbf{E}_1|^2$$

$$r^2 = |b_0 \mathbf{E}_0 + b_1 \mathbf{E}_1|^2$$

Subtracting the last equation from the first two and writing the equations as a linear system:

$$\begin{bmatrix} \mathbf{E}_0 \cdot \mathbf{E}_0 & \mathbf{E}_0 \cdot \mathbf{E}_1 \\ \mathbf{E}_1 \cdot \mathbf{E}_0 & \mathbf{E}_1 \cdot \mathbf{E}_1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} \mathbf{E}_0 \cdot \mathbf{E}_0 \\ \mathbf{E}_1 \cdot \mathbf{E}_1 \end{bmatrix}$$

The system is solved for b_0 and b_1 , and then $b_2 = 1 - b_0 - b_1$. One of the equations for r^2 may be evaluated to produce the radius r .

If you were to compute the circle that passes through all three points that are shown in Figure 13.1 (b), it would have a different center than the minimum-area circle, and its radius would be larger than that of the minimum-area circle. You would also then compute the circles supported by the three combinations of pairs of points. If any such circle contained the third point, it would be a candidate for the minimum-area circle. Thus, you have to compute at most four circles, each containing the input points, and the function should output that circle of minimum area. Rather than implement `MinimumSphere3` to do this work, the logic already exists in the update of the support set to compare all possible combinations of two, three, and four points. Specifically, it will compute spheres supported by two points, so there is no need for `MinimumSphere3` to test pairs of points. It is sufficient to have functions `ExactSphere1`, `ExactSphere2`, `ExactSphere3`, and `ExactSphere4`. Any input points to these that are degenerate (three collinear points, four coplanar points or collinear points) will be handled elsewhere in the program logic.

As explained previously, the function `ExactSphere3` computes the circle passing through three noncollinear points. The function `ExactSphere4` computes the sphere passing through four noncoplanar points. You should recognize this as the problem

of computing a circumscribed sphere for a tetrahedron. Let $\mathbf{S}_i$ be the input points for $0 \leq i \leq 3$. The center of the sphere containing these points has a center written in barycentric coordinates as

$$\mathbf{C} = b_0\mathbf{S}_0 + b_1\mathbf{S}_1 + b_2\mathbf{S}_2 + b_3\mathbf{S}_3$$

where $b_i \in [0, 1]$ and $b_0 + b_1 + b_2 + b_3 = 1$. The center is equidistant from the three points, so

$$|\mathbf{C} - \mathbf{S}_0| = |\mathbf{C} - \mathbf{S}_1| = |\mathbf{C} - \mathbf{S}_2| = |\mathbf{C} - \mathbf{S}_3| = r^2$$

where r is the (as yet unknown) radius of the sphere. From these conditions, we have

$$\mathbf{C} - \mathbf{S}_0 = b_0\mathbf{E}_0 + b_1\mathbf{E}_1 + b_2\mathbf{E}_2 - \mathbf{E}_0$$

$$\mathbf{C} - \mathbf{S}_1 = b_0\mathbf{E}_0 + b_1\mathbf{E}_1 + b_2\mathbf{E}_2 - \mathbf{E}_1$$

$$\mathbf{C} - \mathbf{S}_2 = b_0\mathbf{E}_0 + b_1\mathbf{E}_1 + b_2\mathbf{E}_2 - \mathbf{E}_2$$

$$\mathbf{C} - \mathbf{S}_3 = b_0\mathbf{E}_0 + b_1\mathbf{E}_1 + b_2\mathbf{E}_2$$

where $\mathbf{E}_i = \mathbf{S}_i - \mathbf{S}_3$ for $0 \leq i \leq 2$. These lead to

$$r^2 = |b_0\mathbf{E}_0 + b_1\mathbf{E}_1 + b_2\mathbf{E}_2|^2 - 2\mathbf{E}_0 \cdot (b_0\mathbf{E}_0 + b_1\mathbf{E}_1 + b_2\mathbf{E}_2) + |\mathbf{E}_0|^2$$

$$r^2 = |b_0\mathbf{E}_0 + b_1\mathbf{E}_1 + b_2\mathbf{E}_2|^2 - 2\mathbf{E}_1 \cdot (b_0\mathbf{E}_0 + b_1\mathbf{E}_1 + b_2\mathbf{E}_2) + |\mathbf{E}_1|^2$$

$$r^2 = |b_0\mathbf{E}_0 + b_1\mathbf{E}_1 + b_2\mathbf{E}_2|^2 - 2\mathbf{E}_2 \cdot (b_0\mathbf{E}_0 + b_1\mathbf{E}_1 + b_2\mathbf{E}_2) + |\mathbf{E}_2|^2$$

$$r^2 = |b_0\mathbf{E}_0 + b_1\mathbf{E}_1 + b_2\mathbf{E}_2|^2$$

Subtracting the last equation from the first three and writing the equations as a linear system:

$$\begin{bmatrix} \mathbf{E}_0 \cdot \mathbf{E}_0 & \mathbf{E}_0 \cdot \mathbf{E}_1 & \mathbf{E}_0 \cdot \mathbf{E}_2 \\ \mathbf{E}_1 \cdot \mathbf{E}_0 & \mathbf{E}_1 \cdot \mathbf{E}_1 & \mathbf{E}_1 \cdot \mathbf{E}_2 \\ \mathbf{E}_2 \cdot \mathbf{E}_0 & \mathbf{E}_2 \cdot \mathbf{E}_1 & \mathbf{E}_2 \cdot \mathbf{E}_2 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} \mathbf{E}_0 \cdot \mathbf{E}_0 \\ \mathbf{E}_1 \cdot \mathbf{E}_1 \\ \mathbf{E}_2 \cdot \mathbf{E}_2 \end{bmatrix}$$

The system is solved for b_0 , b_1 , and b_2 , and then $b_3 = 1 - b_0 - b_1 - b_2$. One of the equations for r^2 may be evaluated to produce the radius r .

An obvious question about the original pseudocode is, How often does the algorithm restart itself by setting $i = 0$? You might think restarts are frequent, but as shown in [Wel91], they do not occur often when the input points are randomly permuted. Intuitively, once the bounding sphere of the support set is sufficiently large, most of the points must lie in this sphere, so you rarely restart.

13.1.3 MERGING SPHERES

The algorithm described here computes the smallest sphere containing two spheres. Let the spheres S_i be $|\mathbf{X} - \mathbf{C}_i|^2 = r_i^2$ for $i = 0, 1$. Define $L = |\mathbf{C}_1 - \mathbf{C}_0|$ and unit-length vector $\mathbf{U} = (\mathbf{C}_1 - \mathbf{C}_0)/L$. The problem can be reduced to one dimension by projecting the spheres onto the line $\mathbf{C}_0 + t\mathbf{U}$. The projected intervals in terms of parameter t are $[-r_0, r_0]$ for S_0 and $[L - r_1, L + r_1]$ for S_1 .

If $[-r_0, r_0] \subseteq [L - r_1, L + r_1]$, then $S_0 \subseteq S_1$ and the two spheres merge into S_1 . The test for this case is $r_0 \leq L + r_1$ and $L - r_1 \leq -r_0$. A single test covers both conditions, $r_1 - r_0 \geq L$. To avoid the square root in computing L , compare instead $r_1 \geq r_0$ and $(r_1 - r_0)^2 \geq L^2$.

If $[L - r_1, L + r_1] \subseteq [-r_0, r_0]$, then $S_1 \subseteq S_0$ and the two spheres merge into S_0 . The test for this case is $L + r_1 \leq r_0$ and $-r_0 \leq L - r_1$. A single test covers both conditions, $r_1 - r_0 \leq -L$. Again to avoid the square root, compare instead $r_1 \leq r_0$ and $(r_1 - r_0)^2 \geq L^2$.

Otherwise, the intervals either have partial overlap or are disjoint. The interval containing the two projected intervals is $[-r_0, L + r_1]$. The corresponding merged sphere whose projection is the containing interval has radius

$$r = \frac{L + r_1 + r_0}{2}$$

The center t -value is $(L + r_1 - r_0)/2$ and corresponds to the point

$$\mathbf{C} = \mathbf{C}_0 + \frac{L + r_1 - r_0}{2} \mathbf{U} = \mathbf{C}_0 + \frac{L + r_1 - r_0}{2L} (\mathbf{C}_1 - \mathbf{C}_0)$$

The pseudocode is

```

Input: Sphere(C0,r0) and Sphere(C1,r1)
centerDiff = C1 - C0;
radiusDiff = r1 - r0;
radiusDiffSqr = radiusDiff * radiusDiff;
Lsqr = centerDiff.SquaredLength();
if (radiusDiffSqr >= Lsqr)
{
    if (radiusDiff >= 0.0f)
    {
        return Sphere(C1,r1);
    }
    else
    {
        return Sphere(C0,r0);
    }
}

```



```

else
{
    L = sqrt(Lsqr);
    t = (L + r1 - r0)/(2 * L);
    return Sphere(C0 + t * centerDiff, (L + r1 + r0)/2);
}

```

For three or more bounding spheres, you can merge the first two spheres into a single sphere, merge the third sphere into this one, and so on for each additional sphere. The implementation is simple enough, but even with three spheres, the resulting sphere is not guaranteed to be the minimum-volume sphere containing the input spheres. Constructing the minimum-volume sphere is nontrivial [FG03] and not something you want to do dynamically at program execution time, because of the computational expense. An alternative is to choose the final bounding sphere so that its center is the average of the input sphere centers and its radius is large enough to contain the input spheres. If there are n spheres with centers $\mathbf{C}_i$ and radii r_i for $0 \leq i < n$, then the final sphere has center

$$\mathbf{C} = \frac{1}{n} \sum_{i=0}^{n-1} \mathbf{C}_i$$

and radius

$$r = \max_{0 \leq i < n} \{ |\mathbf{C}_i - \mathbf{C}| + r_i \}$$

13.2 BOXES

The queries are organized with the point-in-box query first, the computation of bounding volumes second, and the merging of bounding volumes third.

13.2.1 POINT IN BOX

For an axis-aligned box with extreme points $\mathbf{P}_{\min} = (x_0, y_0, z_0)$ and $\mathbf{P}_{\max} = (x_1, y_1, z_1)$, a point $\mathbf{P} = (x, y, z)$ is inside (or on) the box when $x_0 \leq x \leq x_1$, $y_0 \leq y \leq y_1$, and $z_0 \leq z \leq z_1$.

For an oriented box with center $\mathbf{C}$, axis directions $\mathbf{D}_i$, and extents e_i for $0 \leq i \leq 2$, a point $\mathbf{P}$ is inside (or on) the box when

$$|\mathbf{D}_i \cdot (\mathbf{P} - \mathbf{C})| \leq e_i$$

for $0 \leq i \leq 2$. The term inside the absolute value signs is the component of the point relative to the axis $\mathbf{D}_i$ of the coordinate system defined by the box parameters.

13.2.2 BOX CONTAINING POINTS

It is clear how to construct an axis-aligned bounding box that contains a set of n points $P[0]$ through $P[n-1]$:

```
Vector3 min = P[0], max = min;
for (i = 1; i < n; i++)
{
    for (j = 0; j < 3; j++)
    {
        if (P[i][j] < min[j] )
        {
            min[j] = P[i][j];
        }
        else if (P[i][j] > max[j])
        {
            max[j] = P[i][j];
        }
    }
}
```

Five algorithms are presented for computing an oriented bounding box for a set of points. The first algorithm fits an unordered set of points, using the eigenvectors of the covariance matrix as the box axis directions. The second algorithm assumes that the points are vertices of a triangle mesh and that the edges have mass and constant density. The third algorithm also assumes a triangle mesh, but now the triangles have mass and are all constant density. The fourth algorithm assumes a triangle mesh that forms a simple polyhedron (not necessarily convex), and treats the object as a solid mass of constant density. The fifth algorithm shows how to compute the minimum-volume OBB that contains the points.

Fitting Points Using the Mean and Covariance

The goal is to construct an OBB to contain a set of n points, $\mathbf{P}_i$, for $0 \leq i < n$. The center of the OBB is chosen to be the mean (average) of the points,

$$\mathbf{C} = \frac{1}{n} \sum_{j=0}^{n-1} \mathbf{P}_j$$

The axes of the box are selected as unit-length eigenvectors of the covariance matrix

$$\mathbf{M} = \frac{1}{n} \sum_{j=0}^{n-1} (\mathbf{P}_j - \mathbf{C})(\mathbf{P}_j - \mathbf{C})^T$$

If $\mathbf{U}_i$ are unit-length eigenvectors, the extents e_i along those axes are computed from the extreme values of the projections of the points onto those axes:

$$a_i = \min_{0 \leq j < n} \{ \mathbf{U}_i \cdot (\mathbf{V}_j - \mathbf{C}) \}, \quad b_i = \max_{0 \leq j < n} \{ \mathbf{U}_i \cdot (\mathbf{V}_j - \mathbf{C}) \}, \quad e_i = \max\{|a_i|, |b_i|\}$$

This box does not necessarily have the smallest volume of all boxes with the same axis directions. For a tighter fit, adjust the center of the box and choose the extents, as shown, to be

$$\mathbf{C}' = \mathbf{C} + \frac{1}{2} \sum_{i=0}^2 (a_i + b_i) \mathbf{U}_i, \quad e_i = (b_i - a_i)/2$$

The pseudocode is

```
Box3 box; // center, axis[3], extent[3]

// Compute mean of points.
Vector3 sum = P[0];
for (i = 1; i < n; i++)
{
    sum += P[i];
}
box.center = sum/n;

// Compute covariances of points.
Matrix3 mat = 0;
for (i = 0; i < n; i++)
{
    Vector3 diff = P[i] - box.center; // D is n-by-1.
    mat += Tensor(diff,diff); // D*Transpose(D)
}
Matrix3 covariance = mat/n;

// Eigenvectors for covariance matrix are the box axes.
ExtractEigenvectors(covariance,box.axis);

// Compute the extremes of the projection of the points
// onto the box axes.
Vector3 diff = P[0] - box.center;
Vector3 min, max;
for (j = 0; j < 3; j++)
{
    min[j] = Dot(box.axis[j],diff);
    max[j] = min[j];
}
```

```

for (i = 1; i < n; i++)
{
    diff = P[i] - box.center;
    for (j = 0; j < 3; j++)
    {
        float tmp = Dot(box.axis[j], diff);
        if (tmp < min[j])
        {
            min[j] = tmp;
        }
        else if (tmp > max[j])
        {
            max[j] = tmp;
        }
    }
}

// Adjust the box center to be the average of the extremes.
// Also compute the extents.
for (j = 0; j < 3; j++)
{
    box.center += 0.5 * (min[j] + max[j]) * box.axis[j];
    box.extent[j] = 0.5 * (max[j] - min[j]);
}

```

For a vector $\mathbf{W}$, $\text{Tensor}(\mathbf{W}, \mathbf{W})$ is the matrix $\mathbf{W}\mathbf{W}^T$. The code does require an eigensolver for a 3×3 matrix; see Section 16.2.

What I have shown here is the historical approach to computing an OBB that contains the points. Using the mean point $\mathbf{C}$ and covariance matrix $\mathbf{M}$ are how a statistician would view this problem. A physicist, on the other hand, would want to treat the points as point masses (all points having the same mass) and use the principal directions of inertia for the directions of the box axes [Ebe03]. These directions are the eigenvectors of the inertia tensor (mass matrix) $\mathbf{J}$ relative to the center of mass $\mathbf{C}$. It turns out that the covariance matrix and inertia tensor are related by

$$\mathbf{J} = \ell^2 \mathbf{I} - n\mathbf{M}$$

where

$$\ell^2 = \sum_{i=0}^{n-1} |\mathbf{X}_i - \mathbf{C}|^2$$

and $\mathbf{I}$ is the identity matrix. An eigendecomposition for the covariance is $\mathbf{M} = \mathbf{R}\mathbf{D}\mathbf{R}^T$, where $\mathbf{R}$ is a rotation matrix whose columns are eigenvectors of $\mathbf{M}$ and where

D is a diagonal matrix whose diagonal entries are eigenvalues of M . Observe that

$$J = \ell^2 R R^T - n R D R^T = R(\ell^2 I - n D) R^T$$

The matrix $\ell^2 I - n D$ is diagonal, so the rotation matrix R also diagonalizes J . That means the eigenspaces for J are the same as those for M , in which case the eigenvectors of the covariance matrix are the principal directions of inertia. It does not matter if you use M or J for the OBB construction!

Fitting Based on Mesh Edges

Assuming the points are vertices of a triangle mesh, the edges of that mesh may be treated as masses of constant density 1. The idea of using edges instead of just vertices is to help avoid skewing of the distribution by localized clusters of points. Points far apart, but connected by an edge, contribute more significantly to the distribution because of the mass of the edge connecting them.

Suppose there are k edges in the mesh. If the i th edge has vertices $\mathbf{P}_{i,0}$ and $\mathbf{P}_{i,1}$, then the edge and its interior are represented by

$$\mathbf{X}_i(s) = \mathbf{P}_{i,0} + s(\mathbf{P}_{i,1} - \mathbf{P}_{i,0})$$

where $s \in [0, 1]$. The mass of the edge is formulated generally as an integral. The mass differential is $dm = L_i ds$, where

$$L_i = |\mathbf{P}_{i,1} - \mathbf{P}_{i,0}|$$

is the length of the edge. I am assuming a constant density of 1. If instead it were a constant δ , then $dm = \delta L_i ds$. The mass of the edge is

$$m_i = \int_0^1 L_i ds = L_i$$

The center of mass of the edge is also formulated generally as an integral

$$\mathbf{C}_i = \frac{\int_0^1 \mathbf{X}_i(s) L_i ds}{m_i} = \frac{\mathbf{P}_{i,0} + \mathbf{P}_{i,1}}{2}$$

Observe that the center of mass of the edge is the midpoint of the edge (when you have constant density).

The total mass of all the edges is

$$m = \sum_{i=1}^k m_i = \sum_{i=1}^k L_i$$

and the center of mass of all the edges is

$$\mathbf{C} = \frac{\sum_{i=1}^k \int_0^1 \mathbf{X}_i(s) L_i ds}{\sum_{i=1}^k \int_0^1 L_i ds} = \sum_{i=1}^k \frac{L_i}{m} \mathbf{C}_i = \sum_{i=1}^k w_i \mathbf{C}_i$$

where $w_i = L_i/m$ and where $\sum_{i=1}^k w_i = 1$. Thus, the center of mass for all edges is a weighted average of the centers of mass for the individual edges.

A scaled covariance matrix for the edge masses is

$$\begin{aligned} \mathbf{M} &= \frac{\sum_{i=1}^k \int_0^1 (\mathbf{X}_i(s) - \mathbf{C})(\mathbf{X}_i(s) - \mathbf{C})^T L_i ds}{\sum_{i=1}^k \int_0^1 L_i ds} \\ &= \frac{1}{6} \sum_{i=1}^k \frac{L_i}{m} \left([\mathbf{D}_{i,0} \quad \mathbf{D}_{i,1}] \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} [\mathbf{D}_{i,0} \quad \mathbf{D}_{i,1}]^T \right) \end{aligned}$$

where $\mathbf{D}_{i,j} = \mathbf{P}_{i,j} - \mathbf{C}$ and is treated as a column vector for the purpose of the matrix multiplications in the expression. The scaling of the covariance matrix makes the measurement dimensionless, which helps with the robustness of the floating-point calculations, and it allows you to reuse the weights $w_i = L_i/m$. The eigenvectors of $\mathbf{M}$ are used as the directions of the box axes and the center of mass $\mathbf{C}$ is adjusted to $\mathbf{C}'$, as was the case when fitting a collection of point masses.

Fitting Based on Mesh Faces

The idea of treating edges in a triangle mesh as having mass may be extended to treating the triangles themselves as having mass.

Suppose there are k triangles in the mesh. If the i th triangle has vertices $\mathbf{P}_{i,j}$ for $0 \leq j \leq 2$, then the triangle and its interior are represented by

$$\mathbf{X}_i(s, t) = \mathbf{P}_{i,0} + s(\mathbf{P}_{i,1} - \mathbf{P}_{i,0}) + t(\mathbf{P}_{i,2} - \mathbf{P}_{i,0})$$

where $s \geq 0, t \geq 0$, and $s + t \leq 1$. The mass of the triangle is formulated generally as an integral. The mass differential is $dm = A_i ds dt$, where

$$A_i = |(\mathbf{P}_{i,1} - \mathbf{P}_{i,0}) \times (\mathbf{P}_{i,2} - \mathbf{P}_{i,0})|$$

is the area of the parallelogram related to the two edges of the triangle. The mass of the triangle is

$$m_i = \int_0^1 \int_0^{1-t} A_i ds dt = \frac{A_i}{2}$$

The center of mass is also formulated generally as an integral

$$\mathbf{C}_i = \frac{\int_0^1 \int_0^{1-t} \mathbf{X}_i(s, t) A_i ds dt}{m_i} = \frac{\mathbf{P}_{i,0} + \mathbf{P}_{i,1} + \mathbf{P}_{i,2}}{3}$$

which is the average of the vertices. This point is also called the *centroid* of the triangle.

The total mass of all the triangles is

$$m = \sum_{i=1}^k m_i = \frac{1}{2} \sum_{i=1}^k A_i$$

and the center of mass of all the triangles is

$$\mathbf{C} = \frac{\sum_{i=1}^k \int_0^1 \int_0^{1-t} \mathbf{X}_i(s, t) A_i ds dt}{\sum_{i=1}^k \int_0^1 \int_0^{1-t} A_i ds dt} = \sum_{i=1}^k \frac{A_i}{2m} \mathbf{C}_i = \sum_{i=1}^k w_i \mathbf{C}_i$$

where $w_i = A_i/(2m)$ and where $\sum_{i=1}^k w_i = 1$. Thus, the center of mass for all triangles is a weighted average of the centers of mass for the individual triangles.

A scaled covariance matrix for the triangle masses is

$$\begin{aligned} \mathbf{M} &= \frac{\sum_{i=1}^k \int_0^1 \int_0^{1-t} (\mathbf{X}_i(s, t) - \mathbf{C})(\mathbf{X}_i(s, t) - \mathbf{C})^T A_i ds dt}{\sum_{i=1}^k \int_0^1 \int_0^{1-t} A_i ds dt} \\ &= \frac{1}{24} \sum_{i=1}^k \frac{A_i}{2m} \left(\begin{bmatrix} \mathbf{D}_{i,0} & \mathbf{D}_{i,1} & \mathbf{D}_{i,2} \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} \mathbf{D}_{i,0} & \mathbf{D}_{i,1} & \mathbf{D}_{i,2} \end{bmatrix}^T \right) \end{aligned}$$

where $\mathbf{D}_{i,j} = \mathbf{P}_{i,j} - \mathbf{C}$ and is treated as a column vector for the purpose of the matrix multiplications in the expression. The scaling of the covariance matrix makes the measurement dimensionless, which helps with the robustness of the floating-point calculations, and it allows you to reuse the weights $w_i = A_i/(2m)$. The eigenvectors of $\mathbf{M}$ are used as the directions of the box axes and the center of mass $\mathbf{C}$ is adjusted to $\mathbf{C}'$, as was the case when fitting a collection of point masses.

Fitting Based on a Mesh Solid

We have fitted a mesh as point masses, edge masses, and face masses. The next step is to fit a mesh as a solid, assuming that the mesh forms a simple polyhedron. The center of mass is $\mathbf{C}$ and the inertia tensor is $\mathbf{J}$. The eigenvectors of $\mathbf{J}$ are used as the directions of the box axes, and the box center is the adjusted point $\mathbf{C}'$ that we saw in the discussion for fitting point masses.

A fast computation of $\mathbf{C}$ and $\mathbf{J}$ was first shown in [Mir96] and uses the Divergence theorem and Green's theorem from calculus. The expressions in that paper have

hidden subexpressions that can be factored out, as was shown in [Ebe03], and leads to even faster computation but requires that the mesh faces be triangles. Using basic mathematics, an extension was made by [Kal] to handle polygon faces, but I will not discuss this algorithm here.¹ The implementation of the algorithm in [Ebe03] is found in the files named `Wm4PolyhedralMassProperties`. Mirtich's implementation of the algorithm in [Mir96] is available online. It is simple enough to set up an experiment to compare the execution times.

Minimum-Volume Box

Naturally, a good candidate for the best-fitting bounding box of a set of points is the OBB of minimum volume of all those OBBs that contain the points. The construction of such a box is an extension of a similar idea in two dimensions: Compute the minimum-area oriented rectangle that contains a set of points. In the 2D problem, observe that any rectangle containing the points necessarily contains the convex hull of the points. In fact, if any points touch the rectangle, they must be vertices of the convex hull. It is sufficient then to compute the convex hull, which is a convex polygon, and solve the problem of computing the minimum-area oriented rectangle that contains a convex polygon. This problem was solved in [Tou84], and the approach is referred to as the *method of rotating calipers*.

The result is that at least one edge of the minimum-area rectangle must be coincident with an edge of the polygon. A simple algorithm to compute the rectangle is to iterate over the edges of the polygon. For each edge, compute the bounding rectangle with the orientation implied by that edge. Of all such rectangles, choose the one with smallest area. For n vertices, this approach is $O(n^2)$ since for each of n edges, you must project n vertices onto an axis in the edge direction and onto an axis perpendicular to the edge direction. The method of rotating calipers, though, shows that you can compute the rectangle in $O(n)$ time. You start with one edge of the polygon and compute the bounding rectangle with the implied orientation. You then rotate the rectangle to the orientation implied by the next edge, but instead of projecting all vertices, you can keep track of the extreme vertices for each orientation and update them as you rotate for each edge. The updates are done in $O(1)$ time for each of n edges, so the final time is $O(n)$.

In three dimensions, it is also sufficient to construct a bounding box for the convex hull of the input points, which is a convex polyhedron. The paper [O'R85] shows that the minimum-volume OBB for a convex polyhedron must (1) have a face that is coincident with a face of the polyhedron or (2) be supported by three mutually perpendicular edges of the polyhedron. For a polyhedron of n vertices, this

1. The author has chosen to apply for a patent on his algorithm. The trend for obtaining patents on software and mathematical algorithms is quite disturbing, especially when the innovation is minor, and even more so when anyone with undergraduate training in mathematics can discover the same results through deductive logic.

can be done in $O(n^3)$ time. The construction has two phases. First, you iterate over the faces of the polyhedron. For each face, project the polyhedron onto the plane of that face. The projection is a convex polygon. The projection of a tight-fitting OBB with a face coincident to the polyhedron face is a tight-fitting oriented rectangle for the convex polygon. This is now the 2D problem of computing the minimum-area rectangle that contains a convex polygon. Use the method of rotating calipers, which determines the orientation of the OBB about the normal to the projection plane. The third dimension of the box is determined by the projection of the convex polyhedron onto the normal line for the face. For each of the $O(n)$ faces, you need $O(n)$ time to compute the smallest OBB with the orientation implied by the face for a total time of $O(n^2)$. The second phase is to iterate over all triples of the $O(n)$ edges. Each iteration tests if the three edges are mutually perpendicular. If they are, their directions are used as box-axis directions. The smallest OBB with this orientation is computed. This phase takes $O(n^3)$ time. The OBB with smallest volume is chosen from all those boxes computed in the two phases.

If the bounding boxes are to be used for culling purposes, and if the objects change dynamically (vertex morphing) or if a parent's bounding box changes dynamically to fit its children's bounding boxes, the minimum-volume box algorithm is most likely too computationally intensive for real-time applications. Generally, a lesser fit is used in order to dynamically update faster, a trade-off of accuracy for speed.

13.2.3 MERGING BOXES

If two oriented boxes were built to contain two separate sets of data points, it is possible to build a single oriented bounding box that contains the union of the sets. That box might not contain the two original oriented boxes, but this is not a problem for culling purposes. All that matters is that the box containing the union of sets is a bounding box. What can be a problem is the time it takes to build the single oriented box.

An alternate approach is to construct an oriented box from only the original boxes that, in addition, contains the original boxes. This can be done by interpolation of the box centers and axes, followed by growing the box to contain the originals. The axes of a box may be stored as the columns of a rotation matrix. This matrix can then be represented by a quaternion. Let the two quaternions for the boxes be q_0 and q_1 with $q_0 \cdot q_1 \geq 0$. The dot product is computed treating the quaternions as 4-tuples. An orientation for the box that bounds the two boxes is chosen to be the one associated with the quaternion $q = (q_0 + q_1)/|q_0 + q_1|$. The absolute value signs indicate length of a 4-tuple. A rotation matrix is computed from q and the columns are used as the directions of the final box axes. Given this orientation, the eight vertices from the two original boxes are projected onto the axes. The box center has components chosen to be the midpoints of the intervals of projection. The pseudocode is

```

// Box has center, axis[3], extent[3].
Input:  Box box0, Box box1
Output: Box box

// Initial center C to use for the projections. This
// will be updated later to C' based on the intervals
// of projection.
box.center = (box0.center + box1.center)/2;

// Compute axes.
Quaternion q0 = ConvertAxesToQuaternion(box0.axis);
Quaternion q1 = ConvertAxesToQuaternion(box1.axis);
if (Dot(q0,q1) < 0)
{
    q1 = -q1;
}
Quaternion q = q0 + q1;
q /= Length(q);
box.axis = ConvertQuaternionToAxes(q);

// Compute projections.
Vector3 min(0,0,0), max = min, diff;
for each vertex V of box0 and box1 do
{
    diff = V - box.center;
    for (j = 0; j < 3; j++)
    {
        if (diff[j] < min[j])
        {
            min[j] = diff[j];
        }
        else if (diff[j] > max[j])
        {
            max[j] = diff[j];
        }
    }
}

// Adjust center and compute extents.
for (j = 0; j < 3; j++)
{
    box.center += 0.5 * (min[j] + max[j]) * box.axis[j];
    box.extent[j] = 0.5 * (max[j] - min[j]);
}

```

The function `ConvertAxesToQuaternion` stores the axes as columns of a rotation matrix, then uses the algorithm to convert a rotation matrix to a quaternion. The function `ConvertQuaternionToAxes` converts the quaternion to a rotation matrix, then extracts the axes as columns of the matrix.

The merging of multiple boxes two at a time suffers from the same problem we saw for bounding spheres. The final box becomes much larger than what it needs to be. Alternatively, you can use the ideas for merging two boxes to merge n boxes. Let q_i for $0 \leq i < n$ be quaternions that represent the orientations of the boxes. Change a quaternion's sign, when necessary, to guarantee that the w -component is nonnegative. This arranges for all the quaternions to be on one hemisphere of the unit hypersphere in four dimensions. Compute the normalized average:²

$$q = \frac{\sum_{i=0}^{n-1} q_i}{|\sum_{i=0}^{n-1} q_i|}$$

Start with a center point that is the average of the input box center points. Compute the final box axes from this quaternion; project all the box vertices onto the axes (containing the current center point); and update the center point and compute the extents as was shown in the pseudocode.

13.3 CAPSULES

The queries are organized with the point-in-capsule query first, the computation of bounding volumes second, and the merging of bounding volumes third.

13.3.1 POINT IN CAPSULE

Given a capsule whose medial set M is a line segment and whose radius is $r > 0$, a point $\mathbf{P}$ is contained in the capsule when

$$\text{Distance}(\mathbf{P}, M) \leq r$$

The distance between a point and a line segment is discussed in Section 14.1.3.

2. Even though all the quaternions are in the same hemisphere, it is still possible for the average to be zero. You must guard against this. A better approach, but one requiring more computational time, is to compute the minimal cone containing the quaternions and use its axis as the quaternion for the final box. The cone construction is related to computing the minimum-volume sphere containing a collection of points.

13.3.2 CAPSULE CONTAINING POINTS

Two algorithms are presented here for computing a bounding capsule for a set of points.

Least-Squares Fit

Fit the points by a line using the least-squares algorithm described in Section 16.3.2. Let the line be $\mathbf{A} + t\mathbf{W}$, where $\mathbf{W}$ is unit length and $\mathbf{A}$ is the average of the points. This line will contain the capsule line segment. Compute r to be the maximum distance from the data points to the line. Select unit vectors $\mathbf{U}$ and $\mathbf{V}$ so that the matrix $R = [\mathbf{U} \ \mathbf{V} \ \mathbf{W}]$, whose columns are the specified vectors, is a rotation matrix. The points can be represented as $\mathbf{X}_i = \mathbf{A} + R\mathbf{Y}_i$, where $\mathbf{Y}_i = (u_i, v_i, w_i)$, but treated as a column vector for the purpose of multiplication by R . In the (u, v, w) coordinate system, the capsule axis is contained by the line $t(0, 0, 1)$. We need to compute the largest ξ_0 so that all points lie above the hemisphere $u^2 + v^2 + (w - \xi_0)^2 = r^2$ with $w \leq \xi_0$. The value is computed as

$$\xi_0 = \min_i \{w_i + \sqrt{r^2 - (u_i^2 + v_i^2)}\}$$

where $0 \leq i \leq n$. Similarly, there is a smallest value ξ_1 so that all points lie below the hemisphere $u^2 + v^2 + (w - \xi_1)^2 = r^2$ with $w \geq \xi_1$. The value is computed as

$$\xi_1 = \max_i \{w_i - \sqrt{r^2 - (u_i^2 + v_i^2)}\}$$

The endpoints of the capsule line segment are $\mathbf{P}_j = \mathbf{A} + \xi_j\mathbf{W}$ for $j = 0, 1$. If, instead, the data points are fit by a least-squares plane $\mathbf{W} \cdot (\mathbf{X} - \mathbf{A}) = 0$, the result is the same, since the unit-length plane normal $\mathbf{W}$ is exactly the line direction.

Minimum of Minimum-Area Projected Circles

For each unit-length direction $\mathbf{W}$ such that $\mathbf{W} \cdot (0, 0, 1) \geq 0$, select unit-length vectors $\mathbf{U}$ and $\mathbf{V}$ so that the matrix $R = [\mathbf{U} \ \mathbf{V} \ \mathbf{W}]$, whose columns are the specified vectors, is a rotation matrix. The points can be represented as $\mathbf{X}_i = \mathbf{A} + R\mathbf{Y}_i$, where $\mathbf{Y}_i = (u_i, v_i, w_i)$, but treated as a column vector for the purpose of multiplication by R . The projections of the points onto the plane $\mathbf{W} \cdot \mathbf{X} = 0$ are (u_i, v_i) . The minimum-area circle containing these points can be computed; say, the radius is $r = r(\mathbf{W})$ and the center is $\mathbf{C} = \mathbf{C}(\mathbf{W})$, the notation indicating that the radius and center are functions of $\mathbf{W}$. Compute the vector $\mathbf{W}'$ that minimizes $r(\mathbf{W})$. The capsule radius is $r(\mathbf{W}')$, and let $w_{\min}$ and $w_{\max}$ be the extreme values for the w_i . The capsule line segment has endpoints $\mathbf{P}_0 = \mathbf{C}(\mathbf{W}') + w_{\min}\mathbf{W}'$ and $\mathbf{P}_1 = \mathbf{C}(\mathbf{W}') + w_{\max}\mathbf{W}'$.

13.3.3 MERGING CAPSULES

If one capsule contains the other, just use the containing capsule. To determine if this is the case, it is simple to formulate a test to see if a sphere of radius r_s and center $\mathbf{C}$ is contained in a capsule with medial segment M and radius r_c . Let d be the distance from $\mathbf{C}$ to M (distance from point to segment). The sphere is inside the capsule as long as $d + r_s \leq r_c$. If $\mathbf{C}$ is a point on M , then $d = 0$ and $r_s \leq r_c$ is an obvious condition for the sphere being inside the capsule. If $\mathbf{C}$ is not on M , let $\mathbf{K}$ be the closest point on the segment to $\mathbf{C}$. The extreme point of the sphere relative to the segment is $\mathbf{C} + r_s(\mathbf{C} - \mathbf{K})/|\mathbf{C} - \mathbf{K}|$ and is a distance $d + r_s$ from M . To test if one capsule is contained by another, it is sufficient to test if the two end spheres of the one capsule are contained in the other capsule. Because the capsules are convex, the containment of the end spheres guarantees that the remainder of the capsule is contained.

When one capsule does not contain the other, let the capsules have radii $r_i > 0$, and let their segments have centers $\mathbf{C}_i$, directions $\mathbf{D}_i$, and extents e_i for $i = 0, 1$. If $\mathbf{D}_0 \cdot \mathbf{D}_1 < 0$, replace $\mathbf{D}_1$ by $-\mathbf{D}_1$. The segment endpoints are $\mathbf{C}_i \pm e_i \mathbf{D}_i$.

The line L containing the final capsule segment is chosen to have origin $\mathbf{C} = (\mathbf{C}_0 + \mathbf{C}_1)/2$, which is the average of the centers of the capsules. This point, however, will not be the final capsule segment's midpoint. The direction vector of the line is obtained by averaging the unit direction vectors of the input capsules, $\mathbf{D} = (\mathbf{D}_0 + \mathbf{D}_1)/|\mathbf{D}_0 + \mathbf{D}_1|$. The final capsule radius r must be chosen sufficiently large so that the final capsule contains the original capsules. It is enough to consider the spherical ends of the original capsules. The endpoints of the capsule segments are $\mathbf{P}_{ij} = \mathbf{C}_i + (2j - 1)e_i \mathbf{D}_i$ for $i = 0, 1$ and $j = 0, 1$. The final radius is

$$r = \max_{i,j} \{\text{Distance}(\mathbf{P}_{ij}, L) + r_i\}$$

Observe that $r \geq r_i$ for $i = 0, 1$. Figure 13.2 illustrates these quantities. The goal now is to choose the endpoints $\mathbf{E}_0$ and $\mathbf{E}_1$ of the final capsule segment. These are of the form $\mathbf{E}_i = \mathbf{C} + t_i \mathbf{D}$ and must be chosen so that the final capsule contains the input capsules. From the illustration, it should be clear that the distance from $\mathbf{E}_0$ to a capsule segment endpoint plus the capsule radius must be smaller or equal to the final capsule radius. That is,

$$|\mathbf{E}_0 - (\mathbf{C}_0 - e_0 \mathbf{D}_0)| + r_0 \leq r \quad \text{and} \quad |\mathbf{E}_0 - (\mathbf{C}_1 - e_1 \mathbf{D}_1)| + r_1 \leq r$$

Define $\Delta_i = \mathbf{C} - (\mathbf{C}_i - e_i \mathbf{D}_i)$. Solve the equalities, which when squared are quadratic equations:

$$t^2 + 2(\mathbf{D} \cdot \Delta_i)t + |\Delta_i|^2 - (r - r_i)^2 = 0$$

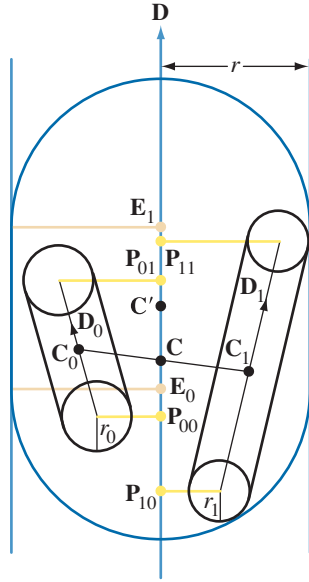


Figure 13.2 Merging two capsules.

Compute the roots of the two equations and choose t_0 to be the smallest of them. Similarly,

$$|E_1 - (C_0 + e_0 D_0)| + r_0 \leq r \quad \text{and} \quad |E_1 - (C_1 + e_1 D_1)| + r_1 \leq r$$

Define $\delta_i = C - (C_i + e_i D_i)$. Solve the equalities, which when squared are quadratic equations:

$$t^2 + 2(D \cdot \delta_i)t + |\delta_i|^2 - (r - r_i)^2 = 0$$

Compute the roots of the two equations and choose t_1 to be the largest of them. Once E_0 and E_1 have been computed, choose the final capsule segment center to be $C' = (E_0 + E_1)/2$.

13.4 LOZENGES

The queries are organized with the point-in-lozenge query first, the computation of bounding volumes second, and the merging of bounding volumes third.

13.4.1 POINT IN LOZENGE

Given a lozenge whose medial set M is a rectangle and whose radius is $r > 0$, a point P is contained in the capsule when

$$\text{Distance}(P, M) \leq r$$

The distance between a point and a rectangle is discussed in Section 14.5.

13.4.2 LOZENGE CONTAINING POINTS

Two algorithms are presented here for computing a bounding lozenge for a set of points.

Fitting Points Using the Mean and Covariance

Compute the mean A of the points and compute the covariance matrix, just as in the algorithm for fitting with an oriented box. Let unit-length eigenvectors of the matrix be U , V , and W . Assume these are labeled so that U corresponds to the largest eigenvalue and W corresponds to the smallest eigenvalue. The data points are represented as $X_i = A + u_i U_i + v_i V_i + w_i W_i$. Let $w_{\min}$ and $w_{\max}$ be the extreme values for the w_i . The data points are bounded by the two planes $W \cdot (X - A) = w_{\min}$ and $W \cdot (X - A) = w_{\max}$. Set the lozenge radius to $r = (w_{\max} - w_{\min})/2$ and adjust the mean to $A \leftarrow A + ((w_{\max} + w_{\min})/2)W$.

Analogous to the fitting of data by a 3D capsule, construct a 2D capsule containing the pairs (w_i, v_i) . We need to compute the largest β_0 so that all points lie above the semicircle $w^2 + (v - \beta_0)^2 = r^2$ with $v \leq \beta_0$. The value is computed as

$$\beta_0 = \min_i \left\{ v_i + \sqrt{r^2 - w_i^2} \right\}$$

where $0 \leq i \leq n$. Similarly, there is a smallest value β_1 so that all points lie below the semicircle $w^2 + (v - \beta_1)^2 = r^2$ with $v \geq \beta_1$. The value is computed as

$$\beta_1 = \max_i \left\{ v_i - \sqrt{r^2 - w_i^2} \right\}$$

The endpoints of the projected capsule line segment determine an edge of the lozenge, $E_1 = (\beta_1 - \beta_0)V$.

Repeat this process for the pairs (u_i, w_i) to obtain values

$$\alpha_0 = \min_i \left\{ u_i + \sqrt{r^2 - w_i^2} \right\}$$

and

$$\alpha_1 = \max_i \left\{ u_i - \sqrt{r^2 - w_i^2} \right\}$$

Although it appears that the other lozenge edge should be $\mathbf{E}_0 = (\alpha_1 - \alpha_0)\mathbf{U}$, it might not be. The hemicylinder ends that are attached by the preceding process form mitered corners that enclose more space than the quarter spheres. It is possible for some data points to be inside the hemicylinder overlap, but outside the quarter sphere. The candidate edge $\mathbf{E}_0$ may need to be increased to enclose the outliers.

Let $\mathbf{K}_0 = \mathbf{A} + \alpha_0\mathbf{U} + \beta_0\mathbf{V}$ be one of the corner points of the current lozenge rectangle. Suppose that $\mathbf{P} = \mathbf{A} + \alpha_p\mathbf{U} + \beta_p\mathbf{V} + \gamma_p\mathbf{W}$ is a point outside the quarter sphere centered at $\mathbf{K}$. For this to be true, $|\mathbf{P} - \mathbf{K}_0| > r$. The corner must be adjusted to $\mathbf{K}_1 = \mathbf{A} + \alpha_1\mathbf{U} + \beta_1\mathbf{V}$ so that $|\mathbf{P} - \mathbf{K}_1| = r$. There are two degrees of freedom for the adjustment. One degree is eliminated by requiring $(\alpha'_1, \beta'_1) = t(\alpha_0, \beta_0) + (1 - t)(\alpha_p, \beta_p)$. Replacing in the previous distance equation yields a quadratic in t that can be solved for

$$t = \frac{r^2 - \gamma_p^2}{(\alpha_p - \alpha_0)^2 + (\beta_p - \beta_0)^2}$$

The adjustment on the corner point does not affect previous containment relationships. Thus, the list of input points can be iterated and the corners adjusted as needed.

After the adjustment, the lozenge rectangle parameters are $[\alpha_0, \alpha'_1] \times [\beta_0, \beta'_1]$. The lozenge origin is chosen to be $\mathbf{A} + \alpha_0\mathbf{U} + \beta_0\mathbf{V}$, and the lozenge edges are $\mathbf{E}_0 = (\alpha_1 - \alpha_0)\mathbf{U}$ and $\mathbf{E}_1 = (\beta'_1 - \beta_0)\mathbf{V}$.

Minimization Method

The construction of a lozenge in the previous paragraphs used eigenvectors from the covariance matrix. The same construction can be applied for any choice of orthonormal vectors that form a right-handed system. The corresponding rotation matrices whose columns are the selected vectors form a three-parameter family (the unit quaternions form a three-dimensional manifold in 4-space). Let the parameters be labeled as the 3-tuple $\boldsymbol{\rho}$. The volume for a given set of parameters, $v(\boldsymbol{\rho})$, can be computed by adding the volumes of the pieces forming the lozenge: the rectangular box, the four hemicylinder sides, and the four quarter-sphere corners. A minimization algorithm can be applied to v to obtain parameters $\boldsymbol{\rho}'$ so that $v(\boldsymbol{\rho}')$ is a global minimum.

13.4.3 MERGING LOZENGES

Two lozenges may be merged into a single lozenge that contains them with the following algorithm. This is the direct extension of the algorithm presented for merging two capsules. In the discussion, the index i is 0 or 1 and refers to a particular lozenge. The lozenges have radii $r_i > 0$. Each lozenge rectangle defines a coordinate system. The center $\mathbf{C}_i$ is the origin; the rectangle axis directions $\mathbf{D}_{i0}$ and $\mathbf{D}_{i1}$ are two of the coordinate axis directions; and the unit-length normal to the plane of the rectangle, $\mathbf{N}_i = \mathbf{D}_{i0} \times \mathbf{D}_{i1}$, is the third coordinate axis direction. The extents are e_{i0} .

The plane M containing the final lozenge rectangle is chosen to have the origin $\mathbf{C} = (\mathbf{C}_0 + \mathbf{C}_1)/2$. The plane normal and plane basis vectors are chosen using quaternion averaging, just like we did for OBBs. Let q_i be a quaternion corresponding to the rotation matrix $R_i = [\mathbf{D}_{i0} \ \mathbf{D}_{i1} \ \mathbf{N}_i]$. If $q_0 \cdot q_1 < 0$, replace q_1 by $-q_1$. Compute $q = (q_0 + q_1)/|q_0 + q_1|$ and generate a rotation $R = [\mathbf{D}_0 \ \mathbf{D}_1 \ \mathbf{N}]$ from it. The last column $\mathbf{N}$ is used as the final plane's normal and the first two columns are used as the final plane's basis vectors.

Just as we saw with capsules, the radius of the final lozenge must be large enough so that the spheres occurring at the eight lozenge corners fit within the final lozenge. The sphere centers are

$$\mathbf{P}_{ijk} = \mathbf{C}_i + (2j - 1)e_{i0}\mathbf{D}_{i0} + (2k - 1)e_{i1}\mathbf{D}_{i1}$$

where all three indices have values 0 or 1 (a total of eight possibilities). The final radius may be chosen as

$$r = \max_{i,j,k} \{\text{Distance}(\mathbf{P}_{ijk}, M) + r_i\}$$

where M is the plane of the final rectangle.

In the merging of capsules, the final step was to place hemispherical caps along the line containing the final capsule segment. These caps were chosen to be as close together as possible. The situation with lozenges is slightly more complicated. You now need to place one pair of half-capsule caps along the $\mathbf{D}_0$ axis and one pair along the $\mathbf{D}_1$ axis. In addition to the placement of the half-capsules, you also have control over the lengths of their segments. To place the first pair, use infinite half-cylinders instead, placing them as close together as possible while ensuring that the lozenge corner spheres are contained between them. Their placement determines the segment length for the half-capsules you place in the other direction.

EXERCISE 13.1

I have not provided the mathematical details for the last paragraph of the section, leaving it as an exercise for you to figure out these details and implement the algorithm. ■

13.5 CYLINDERS

The queries are organized with the point-in-cylinder query first, the computation of bounding volumes second, and the merging of bounding volumes third. These are all for finite cylinders.

13.5.1 POINT IN CYLINDER

Let the cylinder axis be $\mathbf{C} + t\mathbf{D}$, where $\mathbf{C}$ is the center of the cylinder, $\mathbf{D}$ is a direction vector, and $|t| \leq h/2$. The cylinder height is h and the cylinder radius is R . The query point is $\mathbf{P}$. Its component in the direction of the cylinder axis is $T = \mathbf{D} \cdot (\mathbf{P} - \mathbf{C})$. If $|T| > h/2$, then $\mathbf{P}$ is outside the cylinder. If $|T| \leq h/2$, it is not yet known whether $\mathbf{P}$ is inside or outside. To be inside, if L is the cylinder axis, we need $\text{Distance}(\mathbf{P}, L) \leq r$. To compute the distance to L , project out the direction component, $\mathbf{Q} = \mathbf{P} - T\mathbf{D}$. The inside condition is then $|\mathbf{Q}| \leq r$.

13.5.2 CYLINDER CONTAINING POINTS

Two algorithms are presented here for computing a bounding cylinder for a set of points. Fit the points by a line using the least-squares algorithm described in Section 16.3.2. Let the line be $\mathbf{A} + t\mathbf{W}$, where $\mathbf{W}$ is unit length and $\mathbf{A}$ is the average of the data points. Select unit vectors $\mathbf{U}$ and $\mathbf{V}$ so that the matrix $R = [\mathbf{U} \ \mathbf{V} \ \mathbf{W}]$ is a rotation matrix. The points can be represented as $\mathbf{X}_i = \mathbf{A} + R\mathbf{Y}_i$, where $\mathbf{Y}_i = (u_i, v_i, w_i)$, but thought of as a column vector for the purpose of multiplication by R .

Least-Squares Line Contains Axis

The cylinder radius is

$$r = \max_i \left\{ \sqrt{u_i^2 + v_i^2} \right\}$$

The cylinder height is $h = w_{\max} - w_{\min}$, where $w_{\min}$ and $w_{\max}$ are the extreme values of the w_i . To conform to the finite cylinder definition, the line must have its translation vector adjusted. The new translation is

$$\mathbf{A}' = \mathbf{A} + \frac{w_{\min} + w_{\max}}{2} \mathbf{W}$$

The line is $\mathbf{A}' + t\mathbf{W}$ and the cylinder is constrained by $|t| \leq h/2$.

13.5.3 LEAST-SQUARES LINE MOVED TO MINIMUM-AREA CENTER

The minimum-area circle containing the (u_i, v_i) values is computed and has center (u', v') and radius r . The least-squares line is shifted to contain the circle center:

$$\mathbf{A}' = \mathbf{A} + u'\mathbf{U} + v'\mathbf{V}$$

The cylinder radius is r and the algorithm in the last subsection is applied to compute h . That algorithm also shifts the line in the direction of $\mathbf{W}$ to $\mathbf{A}'' + t\mathbf{W}$, where

$$\mathbf{A}'' = \mathbf{A}' + \frac{w_{\min} + w_{\max}}{2}\mathbf{W}$$

13.5.4 MERGING CYLINDERS

Let the cylinders have centers $\mathbf{C}_i$, directions $\mathbf{D}_i$, radii r_i , and heights h_i . The line containing the axis of the final cylinder is chosen just like we did for merging capsules. The line origin is the average of cylinder centers, $\mathbf{C} = (\mathbf{C}_0 + \mathbf{C}_1)/2$. The line direction is the average direction, $\mathbf{D} = (\mathbf{D}_0 + \mathbf{D}_1)/|\mathbf{D}_0 + \mathbf{D}_1|$. As before, $\mathbf{D}_1$ is chosen so that $\mathbf{D}_0 \cdot \mathbf{D}_1 \geq 0$.

The idea now is to project the input cylinders onto the line $\mathbf{C} + t\mathbf{D}$, each projection producing an interval. The smallest interval containing the two projected intervals is used to define the final cylinder. Figure 13.3 illustrates. The projection intervals onto the line $t\mathbf{D}$ have endpoints

$$\mathbf{D} \cdot (\mathbf{C}_i - \mathbf{C}) \pm h|\mathbf{D} \cdot \mathbf{D}_i|/2 + r_i\sqrt{1 - (\mathbf{D} \cdot \mathbf{D}_i)^2}$$

The smallest interval covering these is determined by the minimum and maximum of the four endpoints of the projection intervals. This determines the height h of the final cylinder and the final center $\mathbf{C}'$, which is midway between the extremes.

The radius of the final cylinder needs to be computed. Cylinder points are parameterized by

$$\mathbf{X}_i(t, \theta) = \mathbf{C}_i + t\mathbf{D}_i + r_i((\cos \theta)\mathbf{U}_i + (\sin \theta)\mathbf{V}_i)$$

where $\{\mathbf{D}_i, \mathbf{U}_i, \mathbf{V}_i\}$ is an orthonormal set of vectors. Using $\mathbf{C}$ as the origin of a plane with normal $\mathbf{D}$, the projections of the cylinder points onto the plane are

$$\begin{aligned} \mathbf{Y}_i(t, \theta) &= (I - \mathbf{D}\mathbf{D}^T)(\mathbf{X}_i(t, \theta) - \mathbf{C}) \\ &= (I - \mathbf{D}\mathbf{D}^T)(\mathbf{C}_i - \mathbf{C}) + t(I - \mathbf{D}\mathbf{D}^T)\mathbf{D}_i + r_i((\cos \theta)\mathbf{U}'_i + (\sin \theta)\mathbf{V}'_i) \end{aligned}$$

where $\mathbf{U}'_i = (I - \mathbf{D}\mathbf{D}^T)\mathbf{U}_i$ and $\mathbf{V}'_i = (I - \mathbf{D}\mathbf{D}^T)\mathbf{V}_i$. As θ varies, the length of $(\cos \theta)\mathbf{U}'_i + (\sin \theta)\mathbf{V}'_i$ varies. In fact, the points span an ellipse. The vertices of the ellipse

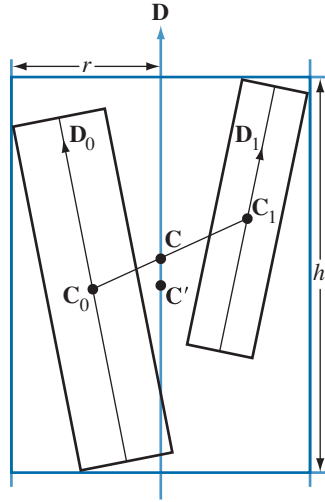


Figure 13.3 Merging of two cylinders.

have the largest magnitude of all such vectors. That magnitude is the maximum of $|\mathbf{U}'_i|$ and $|\mathbf{V}'_i|$. Let $\mathbf{W}_i = \mathbf{U}'_i$ if $|\mathbf{U}'_i|$ is the maximum; otherwise, set $\mathbf{W}_i = \mathbf{V}'_i$. Consequently, the candidates for the projected cylinder points that are farthest from the origin are

$$\left(I - \mathbf{D}\mathbf{D}^T \right) \left((\mathbf{C}_i - \mathbf{C}) \pm \frac{h}{2} \mathbf{D}_0 \pm r \mathbf{W}_i \right)$$

Compute these and choose the final cylinder radius r to be the maximum length.

13.6 ELLIPSOIDS

The queries are organized with the point-in-ellipsoid query first, the computation of bounding volumes second, and the merging of bounding volumes third.

13.6.1 POINT IN ELLIPSOID

For an ellipsoid in standard form, $(x/a)^2 + (y/b)^2 + (z/c)^2 = 1$, a point (x_0, y_0, z_0) is inside (or on) the ellipsoid whenever $(x_0/a)^2 + (y_0/b)^2 + (z_0/c)^2 \leq 1$. For an

ellipsoid in the form $(\mathbf{X} - \mathbf{C})^T \mathbf{M} (\mathbf{X} - \mathbf{C}) = 1$, where $\mathbf{C}$ is the ellipsoid center and $\mathbf{M} = \mathbf{R} \mathbf{D} \mathbf{R}^T$ with $\mathbf{R}$ a rotation matrix and $\mathbf{D}$ a diagonal matrix with positive diagonal entries, a point $\mathbf{P}$ is inside (or on) the ellipsoid whenever

$$(\mathbf{P} - \mathbf{C})^T \mathbf{M} (\mathbf{P} - \mathbf{C}) \leq 1$$

13.6.2 ELLIPSOID CONTAINING POINTS

Two algorithms are presented here for computing a bounding ellipsoid for a set of points. A third algorithm is mentioned, but it is not really one you can use in a real-time situation.

Axis-Aligned Ellipsoid

Given a set of points, a simple way to bound them with an ellipsoid is to first generate the axis-aligned box containing the points and establish the ratios of axis half-lengths. Let $\mathbf{P}_{\min}$ and $\mathbf{P}_{\max}$ be the extreme points for the AABB. The center of the ellipsoid is $\mathbf{C} = (\mathbf{P}_{\max} + \mathbf{P}_{\min})/2$. The half-lengths are components of $\lambda(\mathbf{P}_{\max} - \mathbf{P}_{\min})/2 = \lambda(\delta_0, \delta_1, \delta_2)$, where $\lambda > 0$ is to be determined. Let $\mathbf{D} = \text{Diag}\{1/(\lambda\delta_0)^2, 1/(\lambda\delta_1)^2, 1/(\lambda\delta_2)^2\}$. The bounding ellipsoid is $(\mathbf{X} - \mathbf{C})^T \mathbf{M} (\mathbf{X} - \mathbf{C}) = 1$, where $\mathbf{M} = \mathbf{D} / \max_i \{(\mathbf{P}_i - \mathbf{C})^T \mathbf{D} (\mathbf{P}_i - \mathbf{C})\}$.

Fitting Points with a Gaussian Distribution

This method is similar to the one used for fitting points with an oriented box. The mean of the points is used for the center of the ellipsoid, and the eigenvectors of the covariance matrix are used for the axes. The eigenvalues are used in the same way as the vector $(\delta_0, \delta_1, \delta_2)$ in the fit with an axis-aligned ellipsoid. The ellipsoid is $(\mathbf{X} - \mathbf{C})^T \mathbf{M} (\mathbf{X} - \mathbf{C}) = 1$, where $\mathbf{M} = (\mathbf{R}^T \mathbf{D} \mathbf{R}) / \max_i \{(\mathbf{P}_i - \mathbf{C})^T \mathbf{R}^T \mathbf{D} \mathbf{R} (\mathbf{P}_i - \mathbf{C})\}$.

Minimum-Volume Ellipsoid

While the theory of such a fit has been worked out using randomized linear techniques [Wel91], an implementation is extremely difficult because it requires special-case handlers for computing bounding ellipsoids for sets with up to nine points (the minimum-volume sphere algorithm requires special-case handlers with up to four points). An alternative is to use a constrained numerical minimization, something that is challenging but not impossible to implement. In either case, rapidly computing minimum-volume ellipsoids is not possible at the moment for real-time applications.

13.6.3 MERGING ELLIPSOIDS

Computing a bounding ellipsoid for two other ellipsoids is done in a way similar to that of oriented boxes. The ellipsoid centers are averaged, the quaternions representing the ellipsoid axes are averaged, and then the average is normalized. The original ellipsoids are projected onto the newly constructed axes. On each axis, the smallest interval of the form $[-\sigma, \sigma]$ is computed to contain the intervals of projection. The σ values determine the axis half-lengths for the final ellipsoid.

CHAPTER 14



DISTANCE METHODS

In all but the last section of this chapter, I present a few algorithms for computing distances between points, linear components (lines, rays, segments), triangles, rectangles, and boxes. Linear components are parameterized by $\mathbf{P} + t\mathbf{D}$, where $\mathbf{D}$ is a unit-length vector. The parameter is constrained to $t \in (-\infty, \infty)$ for a line, to $t \in [0, \infty)$ for a ray, and to $t \in [-e, e]$ for a segment. The algorithms are set up to compute squared distance to avoid a potentially expensive square root function call, but if the cost is not of concern to you, your implementation can very well compute distance itself.

To encapsulate the heart of the algorithms, the line-object squared-distance function also returns the parameters for the linear component and for the object. The idea is that the ray-object and segment-object squared-distance functions first call the line-object squared-distance function but then use additional logic to adjust the result based on clamping the parameters to the appropriate intervals. If numerical problems ever occur in a distance query, you need only deal with it (and fix it) in the line-object distance implementations.

The last section of the chapter covers a few miscellaneous algorithms that sometimes arise in computer graphics applications.

14.1 POINT TO LINEAR COMPONENT

The following construction applies in any dimension, not just in three dimensions. Let the query point be $\mathbf{Q}$. The projection onto the line, or onto the line containing the ray or segment, is $\mathbf{K} = \mathbf{P} + \bar{t}\mathbf{D}$, where $\bar{t} = \mathbf{D} \cdot (\mathbf{Q} - \mathbf{P})$.

14.1.1 POINT TO LINE

The squared distance from $\mathbf{Q}$ to the line is

$$\begin{aligned}
 d^2 &= |\mathbf{Q} - \mathbf{K}|^2 \\
 &= |\mathbf{Q} - (\mathbf{P} + \tilde{t}\mathbf{D})|^2 \\
 &= |\mathbf{Q} - \mathbf{P}|^2 - \tilde{t}^2 \\
 &= |\mathbf{Q} - \mathbf{P}|^2 - (\mathbf{D} \cdot (\mathbf{Q} - \mathbf{P}))^2 \\
 &= (\mathbf{Q} - \mathbf{P})^T (I - \mathbf{D}\mathbf{D}^T) (\mathbf{Q} - \mathbf{P}) \\
 &= (\mathbf{Q} - \mathbf{P})^T (I - \mathbf{D}\mathbf{D}^T)^T (I - \mathbf{D}\mathbf{D}^T) (\mathbf{Q} - \mathbf{P}) \\
 &= |(I - \mathbf{D}\mathbf{D}^T)(\mathbf{Q} - \mathbf{P})|^2
 \end{aligned} \tag{14.1}$$

where I is the identity matrix. The last few expressions in the equation indicate that a projection is involved; the matrix $I - \mathbf{D}\mathbf{D}^T$ represents a projection that removes the $\mathbf{D}$ component from vectors, the resulting vectors being perpendicular to $\mathbf{D}$.

The pseudocode for the algorithm is

```

float SquaredDistance (Vector3 Q, Line line, float& tClosest)
{
    Vector3 diff = Q - line.P;
    tClosest = Dot(line.D, diff);
    diff -= tClosest * line.D;
    return diff.SquaredLength();
}

```

14.1.2 POINT TO RAY

When the linear component is a ray, if $\tilde{t} < 0$, the closest point on the ray to $\mathbf{Q}$ is $\mathbf{P}$. For $\tilde{t} \geq 0$, the projection $\mathbf{P} + \tilde{t}\mathbf{Q}$ is the closest point. The squared distance from $\mathbf{Q}$ to the ray is

$$d^2 = \begin{cases} |\mathbf{Q} - \mathbf{P}|^2, & \tilde{t} < 0 \\ |\mathbf{Q} - (\mathbf{P} + \tilde{t}\mathbf{D})|^2, & \tilde{t} \geq 0 \end{cases} \tag{14.2}$$

The pseudocode for the algorithm is

```

float SquaredDistance (Vector3 Q, Ray ray, float& tClosest)
{
    Line line = <convert ray to line>;
    float sqrDistance = SquaredDistance(Q, line, tClosest);
}

```



```

    if (tClosest < 0)
    {
        tClosest = 0;
        Vector3 diff = Q - ray.P;
        sqrDistance = diff.SquaredLength();
    }
    return sqrDistance;
}

```

14.1.3 POINT TO SEGMENT

When the linear component is a segment represented by the center-direction-extent form, if $\bar{t} > e$, then the closest point on the segment to $\mathbf{Q}$ is $\mathbf{P} + e\mathbf{D}$. If $\bar{t} < -e$, then the closest point on the segment to $\mathbf{Q}$ is $\mathbf{P} - e\mathbf{D}$. The squared distance from $\mathbf{Q}$ to the line segment is

$$d^2 = \begin{cases} |\mathbf{Q} - (\mathbf{P} - e\mathbf{D})|^2, & \bar{t} < -e \\ |\mathbf{Q} - (\mathbf{P} + \bar{t}\mathbf{D})|^2, & |\bar{t}| \leq e \\ |\mathbf{Q} - (\mathbf{P} + e\mathbf{D})|^2, & \bar{t} > e \end{cases} \quad (14.3)$$

The pseudocode for the algorithm is

```

float SquaredDistance (Vector3 Q, Segment segment, float& tClosest)
{
    Line line = <convert segment to line>;
    float sqrDistance = SquaredDistance(Q,line,tClosest);
    Vector3 diff;
    if (tClosest < -segment.e)
    {
        tClosest = -segment.e;
        diff = Q - (segment.P - segment.e * segment.D);
        sqrDistance = diff.SquaredLength();
    }
    else if (tClosest > segment.e)
    {
        tClosest = segment.e;
        diff = Q - (segment.P + segment.e * segment.D);
        sqrDistance = diff.SquaredLength();
    }
    return sqrDistance;
}

```

14.2 LINEAR COMPONENT TO LINEAR COMPONENT

This section describes squared-distance algorithms for pairs of linear components.

14.2.1 LINE TO LINE

Two lines are $\mathbf{X}_0(s) = \mathbf{P}_0 + s\mathbf{D}_0$ and $\mathbf{X}_1(t) = \mathbf{P}_1 + t\mathbf{D}_1$. The squared distance between any two points on the line is the quadratic function

$$Q(s, t) = |\mathbf{X}_0(s) - \mathbf{X}_1(t)|^2 = s^2 + 2a_{01}st + t^2 + 2b_0s + 2b_1t + c$$

where $a_{01} = -\mathbf{D}_0 \cdot \mathbf{D}_1$, $b_0 = \mathbf{D}_0 \cdot (\mathbf{P}_0 - \mathbf{P}_1)$, $b_1 = -\mathbf{D}_1 \cdot (\mathbf{P}_0 - \mathbf{P}_1)$, and $c = |\mathbf{P}_0 - \mathbf{P}_1|^2$.

The squared distance between the lines is the minimum squared distance between pairs of points. These points occur at the (s, t) value that minimizes $Q(s, t)$. From calculus, the minimum must occur when the gradient of Q is the zero vector. The gradient is

$$\nabla Q(s, t) = 2(s + a_{01}t + b_0, a_{01}s + t + b_1)$$

Setting this equal to the zero vector produces two equations in the two unknowns s and t :

$$\begin{bmatrix} 1 & a_{01} \\ a_{01} & 1 \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} -b_0 \\ -b_1 \end{bmatrix}$$

The determinant of the coefficient matrix is

$$\delta = 1 - a_{01}^2 = |\mathbf{D}_0 \times \mathbf{D}_1|^2 \geq 0$$

The matrix is invertible when $\delta > 0$. Geometrically, the lines are not parallel and the pair of closest points is unique. Algebraically, we have

$$\begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} 1 & a_{01} \\ a_{01} & 1 \end{bmatrix}^{-1} \begin{bmatrix} -b_0 \\ -b_1 \end{bmatrix} = \frac{1}{\delta} \begin{bmatrix} 1 & -a_{01} \\ -a_{01} & 1 \end{bmatrix} \begin{bmatrix} -b_0 \\ -b_1 \end{bmatrix} = \begin{bmatrix} a_{01}b_1 - b_0 \\ a_{01}b_0 - b_1 \end{bmatrix}$$

If $\delta = 0$, the matrix is not invertible. Geometrically, the lines are parallel, so there are infinitely many pairs of closest points. Algebraically, we have only one independent equation say, $s + a_{01}t = -b_0$. Any pair of closest points may be used to compute the distance, so it is sufficient to choose $s = -b_0$ and $t = 0$.

The pseudocode for the algorithm is

```
float SquaredDistance (Line line0, Line line1, float& l0Closest,
float& l1Closest)
{
```

```

const float epsilon = <small threshold such as 1e-06>;
float sqrDistance;

Vector3 diff = line0.P - line1.P;
float a01 = -Dot(line0.D, line1.D);
float b0 = Dot(line0.D, diff);
float c = diff.SquaredLength();
float det = 1 - a01 * a01;
Real fB1, fS0, fS1, fSqrDist;

if (|det| >= epsilon)
{
    // Lines are not parallel.
    float b1 = -Dot(line1.D, diff);
    float invDet = 1/det;
    l0Closest = (a01 * b1 - b0) * invDet;
    l1Closest = (a01 * b0 - b1) * invDet;
    sqrDistance = l0Closest * (l0Closest + a01 * l1Closest + 2 * b0)
        + l1Closest * (a01 * l0Closest + l1Closest + 2 * b1) + c;
}
else
{
    // Lines are parallel; select any closest pair of points.
    l0Closest = -b0;
    l1Closest = 0;
    sqrDistance = b0 * l0Closest + c;
}

return |sqrDistance|;
}

```

The absolute value call when testing the determinant is used to guard against floating-point round-off errors when the determinant is nearly zero. These round-off errors can lead to a small negative result for the determinant, even though it is theoretically nonnegative. The same problems can occur when computing the squared distance by evaluating the quadratic function, so absolute values are used to guard against this.

14.2.2 LINE TO RAY

Using the encapsulation idea mentioned in the introduction, the line-ray squared-distance calculator uses the line-line squared-distance calculator and then clamps the ray parameter to $[0, \infty)$. The pseudocode is

```

float SquaredDistance (Line line, Ray ray, float& lClosest,
                      float& rClosest)
{
    Line rline = <convert ray to line>;
    float sqrDistance = SquaredDistance(line,rline,lClosest,rClosest);
    if (rClosest < 0)
    {
        rClosest = 0;
        sqrDistance = SquaredDistance(ray.P,line,lClosest);
    }
    return sqrDistance;
}

```

14.2.3 LINE TO SEGMENT

Using the encapsulation idea mentioned in the introduction, the line-segment squared-distance calculator uses the line-line squared-distance calculator and then clamps the segment parameter to $[-e, e]$. The pseudocode is

```

float SquaredDistance (Line line, Segment segment, float& lClosest,
                      float& sClosest)
{
    Line sline = <convert segment to line>;
    float sqrDistance = SquaredDistance(line,sline,lClosest,sClosest);
    Vector3 end;
    if (sClosest < -segment.e)
    {
        sClosest = -segment.e;
        end = segment.P - segment.e * segment.D;
        sqrDistance = SquaredDistance(end,line,lClosest);
    }
    else if (sClosest > segment.e)
    {
        sClosest = segment.e;
        end = segment.P + segment.e * segment.D;
        sqrDistance = SquaredDistance(end,line,lClosest);
    }
    return sqrDistance;
}

```

14.2.4 RAY TO RAY

Using the encapsulation idea mentioned in the introduction, the ray-ray squared-distance calculator uses the line-ray squared-distance calculators and then clamps parameters as needed. The pseudocode is

```
float SquaredDistance (Ray ray0, Ray ray1, float& r0Closest,
    float& r1Closest)
{
    Line line = <convert ray0 to line>;
    float sqrDistance = SquaredDistance(line,ray1,r0Closest,r1Closest);
    if (r0Closest < 0)
    {
        r0Closest = 0;
        sqrDistance = SquaredDistance(ray0.P,ray1,r1Closest);
    }
    return sqrDistance;
}
```

14.2.5 RAY TO SEGMENT

Using the encapsulation idea mentioned in the introduction, the ray-segment squared-distance calculator uses the line-ray squared-distance calculators and then clamps parameters as needed. The pseudocode is

```
float SquaredDistance (Ray ray, Segment segment, float& rClosest,
    float& sClosest)
{
    Line line = <convert ray to line>;
    float sqrDistance = SquaredDistance(line,segment,rClosest,sClosest);
    if (rClosest < 0)
    {
        rClosest = 0;
        sqrDistance = SquaredDistance(ray.P,segment,sClosest);
    }
    return sqrDistance;
}
```

14.2.6 SEGMENT TO SEGMENT

Using the encapsulation idea mentioned in the introduction, the segment-segment squared-distance calculator uses the line-segment squared-distance calculators and then clamps parameters as needed. The pseudocode is

```

float SquaredDistance (Segment segment0, Segment segment1,
    float& s0Closest, float& s1Closest)
{
    Line line = <convert segment0 to line>;
    float sqrDistance = SquaredDistance(line, segment1, s0Closest, s1Closest);
    Vector3 end;
    if (s0Closest < -segment0.e)
    {
        s0Closest = -segment0.e;
        end = segment0.P - segment0.e * segment0.D;
        sqrDistance = SquaredDistance(end, segment1, s1Closest);
    }
    else if (s0Closest > segment0.e)
    {
        s0Closest = segment0.e;
        end = segment0.P + segment0.e * segment0.D;
        sqrDistance = SquaredDistance(end, segment1, s1Closest);
    }
    return sqrDistance;
}

```

14.3 POINT TO TRIANGLE

The problem is to compute the minimum distance between a point $\mathbf{P}$ and a triangle $\mathbf{T}(s, t) = \mathbf{B} + s\mathbf{E}_0 + t\mathbf{E}_1$ for $(s, t) \in D = \{(s, t) : s \in [0, 1], t \in [0, 1], s + t \leq 1\}$. The minimum distance is computed by locating the values $(\bar{s}, \bar{t}) \in D$ corresponding to the point on the triangle closest to $\mathbf{P}$. The squared-distance function for any point on the triangle to $\mathbf{P}$ is $Q(s, t) = |\mathbf{T}(s, t) - \mathbf{P}|^2$ for $(s, t) \in D$. The function is quadratic in s and t :

$$Q(s, t) = a_{00}s^2 + 2a_{01}st + a_{11}t^2 + 2b_0s + 2b_1t + c$$

where $a_{ij} = \mathbf{E}_i \cdot \mathbf{E}_j$, $b_i = \mathbf{E}_i \cdot (\mathbf{B} - \mathbf{P})$, and $c = |\mathbf{B} - \mathbf{P}|^2$. Quadratics are classified by the sign of $a_{00}a_{11} - a_{01}^2$:

$$a_{00}a_{11} - a_{01}^2 = (\mathbf{E}_0 \cdot \mathbf{E}_0)(\mathbf{E}_1 \cdot \mathbf{E}_1) - (\mathbf{E}_0 \cdot \mathbf{E}_1)^2 = |\mathbf{E}_0 \times \mathbf{E}_1|^2 > 0$$

The positivity is based on the assumption that the two edges $\mathbf{E}_0$ and $\mathbf{E}_1$ of the triangle are linearly independent, so their cross product is a nonzero vector. The positivity guarantees that the graph of Q is a paraboloid, and in fact it opens upward since $Q \geq 0$. A paraboloid always has a unique minimum. The goal is to minimize $Q(s, t)$ over D . Since Q is a continuously differentiable function, the minimum occurs either

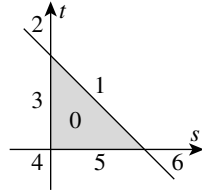


Figure 14.1 Partitioning of the st -plane by triangle domain D .

at an interior point of D where the gradient of Q is the zero vector or at a point on the boundary of D .

The gradient is

$$\nabla Q(s, t) = 2(a_{00}s + a_{01}t + b_0, a_{01}s + a_{11}t + b_1)$$

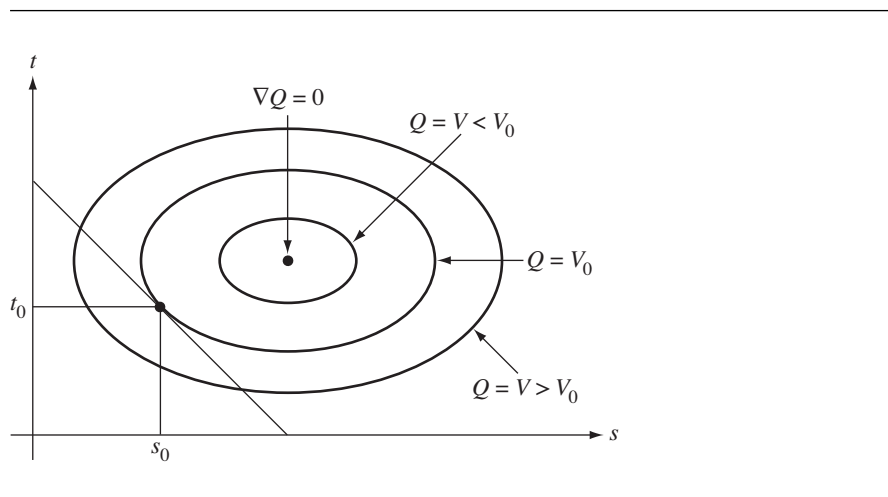
and is zero when

$$\bar{s} = \frac{a_{01}b_1 - a_{11}b_0}{a_{00}a_{11} - a_{01}^2}, \quad \bar{t} = \frac{a_{01}b_0 - a_{00}b_1}{a_{00}a_{11} - a_{01}^2}$$

If $(\bar{s}, \bar{t}) \in D$, then the minimum of Q is found. Otherwise, the minimum must occur on the boundary of the triangle. To find the correct boundary, consider Figure 14.1, which shows a partitioning of the plane implied by the set D . The central triangle labeled region 0 is the domain of Q , $(s, t) \in D$. If $(\bar{s}, \bar{t})$ is in region 0, then the point on the triangle closest to $\mathbf{P}$ is interior to the triangle.

Suppose $(\bar{s}, \bar{t})$ is in region 1. The level curves of Q are those curves in the st -plane for which Q is a constant. Since the graph of Q is a paraboloid, the level curves are ellipses. At the point where $\nabla Q = (0, 0)$, the level curve degenerates to a single point $(\bar{s}, \bar{t})$. The global minimum of Q occurs there; call it $V_{\min}$. As the level values V increase from $V_{\min}$, the corresponding ellipses are increasingly farther away from $(\bar{s}, \bar{t})$. There is a smallest level value V_0 for which the corresponding ellipse (implicitly defined by $Q = V_0$) just touches the triangle domain edge $s + t = 1$ at a value $s = s_0 \in [0, 1]$, $t_0 = 1 - s_0$. For level values $V < V_0$, the corresponding ellipses do not intersect D . For level values $V > V_0$, portions of D lie inside the corresponding ellipses. In particular, any points of intersection of such an ellipse with the edge must have a level value $V > V_0$. Therefore, $Q(s, 1 - s) > Q(s_0, t_0)$ for $s \in [0, 1]$ and $s \neq s_0$. The point (s_0, t_0) provides the minimum squared distance between $\mathbf{P}$ and the triangle. The triangle point is an edge point. Figure 14.2 illustrates the idea by showing various level curves.

An alternate way of visualizing where the minimum distance point occurs on the boundary is to intersect the graph of Q with the plane $s + t = 1$. The curve of

Figure 14.2 Various level curves $Q(s, t) = V$.

intersection is a parabola and is the graph of $F(s) = Q(s, 1 - s)$ for $s \in [0, 1]$. Now the problem has been reduced by one dimension to minimizing a function $F(s)$ for $s \in [0, 1]$. The minimum of $F(s)$ occurs either at an interior point of $[0, 1]$, in which case $F'(s) = 0$ at that point, or at an endpoint $s = 0$ or $s = 1$. Figure 14.2 shows the case when the minimum occurs at an interior point of the edge. At that point the ellipse is tangent to the line $s + t = 1$. In the endpoint cases, the ellipse may just touch one of the vertices of D , but not necessarily tangentially.

To distinguish between the interior point and endpoint cases, the same partitioning idea applies in the 1D case. The interval $[0, 1]$ partitions the real line into three intervals, $s < 0$, $s \in [0, 1]$, and $s > 1$. Let $F'(\hat{s}) = 0$. If $\hat{s} < 0$, then $F(s)$ is an increasing function for $s \in [0, 1]$. The minimum restricted to $[0, 1]$ must occur at $s = 0$, in which case Q attains its minimum at $(s, t) = (0, 1)$. If $\hat{s} > 1$, then $F(s)$ is a decreasing function for $s \in [0, 1]$. The minimum for F occurs at $s = 1$ and the minimum for Q occurs at $(s, t) = (1, 0)$. Otherwise, $\hat{s} \in [0, 1]$, F attains its minimum at $\hat{s}$, and Q attains its minimum at $(s, t) = (\hat{s}, 1 - \hat{s})$.

The occurrence of $(\bar{s}, \bar{t})$ in region 3 or 5 is handled in the same way as when the global minimum is in region 0. If $(\bar{s}, \bar{t})$ is in region 3, then the minimum occurs at $(0, t_0)$ for some $t_0 \in [0, 1]$. If $(\bar{s}, \bar{t})$ is in region 5, then the minimum occurs at $(s_0, 0)$ for some $s_0 \in [0, 1]$. Determining if the first contact point is at an interior or endpoint of the appropriate interval is handled the same as discussed earlier.

If $(\bar{s}, \bar{t})$ is in region 2, it is possible the level curve of Q that provides first contact with the unit square touches either edge $s + t = 1$ or edge $s = 0$. Because the global minimum occurs in region 2, the negative of the gradient at the corner $(0, 1)$ cannot

point inside D . If $\nabla Q = (Q_s, Q_t)$, where Q_s and Q_t are the partial derivatives of Q , it must be that $(0, -1) \cdot \nabla Q(0, 1)$ and $(1, -1) \cdot \nabla Q(0, 1)$ cannot both be negative. The two vectors $(0, -1)$ and $(1, -1)$ are directions for the edges $s = 0$ and $s + t = 1$, respectively. The choice of edge $s + t = 1$ or $s = 0$ can be made based on the signs of $(0, -1) \cdot \nabla Q(0, 1)$ and $(1, -1) \cdot \nabla Q(0, 1)$. The same type of argument applies in region 6. In region 4, the two quantities whose signs determine which edge contains the minimum are $(1, 0) \cdot \nabla Q(0, 0)$ and $(0, 1) \cdot \nabla Q(0, 0)$.

The implementation of the algorithm is designed so that at most one floating-point division is used when computing the minimum distance and corresponding closest points. Moreover, the division is deferred until it is needed, and in some cases no division is needed. Define $\delta = a_{00}a_{11} - a_{01}^2$. In the theoretical development, $\bar{s} = (a_{01}b_1 - a_{11}b_0)/\delta$ and $\bar{t} = (a_{01}b_0 - a_{00}b_1)/\delta$ were computed so that $\nabla Q(\bar{s}, \bar{t}) = (0, 0)$. The location of the global minimum is then tested to see if it is in the triangle domain D . If so, then the information to compute the minimum distance is known. If not, then the boundary of D must be tested. To defer the division by δ , the code instead computes $\bar{s} = a_{01}b_1 - a_{11}b_0$ and $\bar{t} = a_{01}b_0 - a_{00}b_1$ and tests for containment in a scaled domain, $s \in [0, \delta]$, $t \in [0, \delta]$, and $s + t \leq \delta$. If in that set, the divisions are performed. If not, the boundary of the unit square is tested. The general outline of the conditions for determining which region contains $(\bar{s}, \bar{t})$ is

```
det = a00 * a11 - a01 * a01;
s = a01 * b1 - a11 * b0;
t = a01 * b0 - a00 * b1;
if (s + t <= det)
{
    if (s < 0) { if (t < 0) { region 4 } else { region 3 } }
    else if (t < 0) { region 5 }
    else { region 0 }
}
else
{
    if (s < 0) { region 2 }
    else if (t < 0) { region 6 }
    else { region 1 }
}
```

The block of code for handling region 0 is

```
invDet = 1/det;
s *= invDet;
t *= invDet;
```

and requires a single division.

The block of code for region 1 is

```
// F(s) = Q(s,1 - s)
//   = (a00 - 2 * a01 + a11) * s^2 - 2 * (a11 - a01 + b1 - b0) * s
//   + (a11 + 2 * b1 + c)
// F'(s)/2 = (a00 - 2 * a01 + a11) * s + (a01 - a11 + b0 - b1)
// F'(s) = 0 when s = (a11 + b1 - a01 - b0)/(a00 - 2 * a01 + a11)
// a00 - 2 * a01 + a11 = |E0 - E1|^2 > 0, so only the sign of
//   a11 - a01 + b1 - b0 need be considered.

numer = a11 - a01 + b1 - b0;
if (numer <= 0)
{
    s = 0;
}
else
{
    denom = a00 - 2 * a01 + a11; // positive quantity
    s = (numer >= 0 ? 1 : numer/denom);
}
t = 1 - s;
```

The block of code for region 3 is given next. The block of code for region 5 is similar.

```
// F(t) = Q(0,t) = a11 * t^2 + b1 * t + f
// F'(t)/2 = a11 * t + b1
// F'(t) = 0 when t = -b1/a11
s = 0;
t = (b1 >= 0 ? 0 : (-b1 >= a11 ? 1 : -b1/a11));
```

The block of code for region 2 follows. The blocks of code for regions 4 and 6 are similar.

```
// Grad(Q) = 2 * (a00 * s + a01 * t + b0, a01 * s + a11 * t + b1)
// Grad(Q)(0,1) = (a01 + b0, a11 + b1)
// Dot((0,-1), Grad(Q)(0,1)) = -(a11 + b1)
// Dot((1,-1), Grad(Q)(0,1)) = (a01 + b0) - (a11 + b1)
// min on edge s + t = 1 if Dot((1,-1) * Grad(Q)(0,1)) > 0
// min on edge s = 0, otherwise

tmp0 = a01 + b0;
tmp1 = a11 + b1;
if (tmp1 > tmp0) // minimum on edge s + t = 1
```

```

{
    numer = tmp1 - tmp0;
    denom = a00 - 2 * a01 + a11;
    s = (numer >= denom ? 1 : numer/denom);
    t = 1 - s;
}
else // minimum on edge s = 0
{
    s = 0;
    t = (tmp1 <= 0 ? 1 : (b1 >= 0 ? 0 : -b1/a11));
}

```

14.4 LINEAR COMPONENT TO TRIANGLE

The problem is to compute the minimum distance between a linear component $\mathbf{P} + t\mathbf{D}$ and a triangle with vertices $\mathbf{Q}_0$, $\mathbf{Q}_1$, and $\mathbf{Q}_2$. Although it is possible to use the method of constrained quadratic minimization, I will discuss something conceptually simpler.

14.4.1 LINE TO TRIANGLE

There are quite a few approaches you can use to compute the distance between a line and a triangle. All of them must distinguish between the cases of a parallel or nonparallel line and triangle. Let $\mathbf{E}_0 = \mathbf{Q}_1 - \mathbf{Q}_0$ and $\mathbf{E}_1 = \mathbf{Q}_2 - \mathbf{Q}_0$. A unit-length normal vector for the plane of the triangle is

$$\mathbf{N} = \frac{\mathbf{E}_0 \times \mathbf{E}_1}{|\mathbf{E}_0 \times \mathbf{E}_1|}$$

If $\mathbf{N} \cdot \mathbf{D} = 0$, the line and the triangle are parallel. When using floating-point arithmetic, you will want to threshold the value, say, $|\mathbf{N} \cdot \mathbf{D}| \leq \varepsilon$ for a small tolerance ε .

If the line and triangle are not parallel, the intersection point of the line and the plane of the triangle is a solution to

$$\mathbf{P} + t\mathbf{D} = \mathbf{Q}_0 + b_1\mathbf{E}_1 + b_2\mathbf{E}_2$$

Define $b_0 = 1 - b_1 - b_2$. The b_i are barycentric coordinates for the intersection point. If the $b_i \in [0, 1]$, then the intersection point must be inside the triangle, and the distance from the line to the triangle is zero. If any of the b_i is not in $[0, 1]$, then a closest triangle point to the line is on an edge of the triangle. It is sufficient to compute the distance between the line and each edge of the triangle, selecting the minimum of the three values to be the distance between the line and the triangle.

The barycentric coordinates may be computed as follows. Let $\mathbf{U}$ and $\mathbf{V}$ be vectors so that $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{D}$ are mutually perpendicular. Mathematically, it is not necessary for these two vectors to be unit length, but for numerical robustness you might construct them to be unit length. Moreover, if you plan on a lot of line-triangle distance queries with the same line, you can compute the two vectors once and store them for use by the queries. Subtracting $\mathbf{Q}_0$ from both sides of the equation defining the intersection,

$$b_1\mathbf{E}_1 + b_2\mathbf{E}_2 = (\mathbf{P} - \mathbf{Q}_0) + t\mathbf{D}$$

Dot the equation with $\mathbf{U}$ and with $\mathbf{V}$ to obtain two linear equations in the two unknowns b_1 and b_2 . The solution is

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} \mathbf{U} \cdot \mathbf{E}_0 & \mathbf{U} \cdot \mathbf{E}_1 \\ \mathbf{V} \cdot \mathbf{E}_0 & \mathbf{V} \cdot \mathbf{E}_1 \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{U} \cdot (\mathbf{P} - \mathbf{Q}_0) \\ \mathbf{V} \cdot (\mathbf{P} - \mathbf{Q}_0) \end{bmatrix}, \quad b_0 = 1 - b_1 - b_2$$

Also dot the equation with $\mathbf{N}$ and solve for

$$t = \frac{-\mathbf{N} \cdot (\mathbf{P} - \mathbf{Q}_0)}{\mathbf{N} \cdot \mathbf{D}}$$

To avoid the division, dot the equation with $\mathbf{D}$ and solve for

$$t = b_1\mathbf{D} \cdot \mathbf{E}_0 + b_2\mathbf{D} \cdot \mathbf{E}_1 - \mathbf{D} \cdot (\mathbf{P} - \mathbf{Q}_0)$$

The pseudocode for the algorithm is listed next. I assume that the vectors $\mathbf{U}$ and $\mathbf{V}$ have already been computed and are stored with the `Line` data structure. The function returns the squared distance to avoid the square root calculation. It also returns the t -value for the point on the line and the barycentric coordinates b_i for the point on the triangle, both points contributing to the distance calculation.

```
float SquaredDistance (Line line, Triangle triangle,
    float& tClosest, float bClosest[3])
{
    const float epsilon = <small tolerance, say, 1e-06>;
    Vector3 E0 = triangle.Q[1] - triangle.Q[0];
    Vector3 E1 = triangle.Q[2] - triangle.Q[0];
    Vector3 N = UnitCross(E0,E1); // normalized cross product
    float NdD = Dot(N,line.D);
    if (|NdD| > epsilon)
    {
        // The line and triangle are not parallel. Compute
        // the intersection point.
        Vector3 PmQ0 = line.P - triangle.Q[0];
        float UdE0 = Dot(line.U,E0);
```

```

float UdE1 = Dot(line.U,E1);
float UdPmQ0 = Dot(line.U,PmQ0);
float VdE0 = Dot(line.V,E0);
float VdE1 = Dot(line.V,E1);
float VdPmQ0 = Dot(line.V,PmQ0);
float invDet = 1/(UdE0 * VdE1 - UdE1 * VdE0);
float b1 = (VdE1 * UdPmQ0 - UdE1 * VdPmQ0) * invDet;
float b2 = (UdE0 * VdPmQ0 - VdE0 * UdPmQ0) * invDet;
float b0 = 1 - b1 - b2;
if (b0 >= 0 && b1 >= 0 && b2 >= 0)
{
    // The intersection point is inside the triangle.
    tClosest = -Dot(N,PmQ0)/NdD;
    bClosest[0] = b0;
    bClosest[1] = b1;
    bClosest[2] = b2;
    return 0;
}
}

// The line and triangle are not parallel and the line
// does not intersect the triangle. Or the line and
// triangle are parallel.
float sqrDistance = infinity;
Segment seg;
for (i0 = 2, i1 = 0; i1 < 3; i0 = i1++)
{
    seg.P = (triangle.Q[i0] + triangle.Q[i1])/2;
    seg.D = (triangle.Q[i1] - triangle.Q[i0])/2;
    seg.e = Normalize(seg.D)/2;
    float tmpLT, tmpST;
    float tmpSD = SquaredDistance(line,seg,tmpLT,tmpST);
    if (tmpSD < sqrDistance)
    {
        sqrDistance = tmpSD;
        tClosest = tmpLT;
        bClosest[i0] = (1 - tmpST/seg.e)/2;
        bClosest[i1] = 1 - bClosest[i0];
        bClosest[3 - i0 - i1] = 0;
    }
}
}
return sqrDistance;
}

```

The `Normalize` function normalizes its input to unit length and returns the length of the input before normalization.

Just as **U** and **V** may be precomputed and stored with the line's data structure, if the line will be used in multiple distance queries, you may precompute the vectors **E**₀, **E**₁, and **N** and store them with the triangle's data structure if the triangle will be used in multiple distance queries.

14.4.2 RAY TO TRIANGLE

This algorithm may be computed from scratch in a manner similar to that for line-triangle distance, but instead I make use of the line-triangle distance function. The line containing the ray is used in the line-triangle distance function call. One of the return values is t , the parameter corresponding to the line point closest to the triangle. If $t \geq 0$, then this point is on the ray and it is, of course, the closest ray point to the triangle. However, if $t < 0$, then it must be the case that the ray origin is closest to the triangle, and thus you must call a function to compute the distance between a point and a triangle. The pseudocode is

```
float SquaredDistance (Ray ray, Triangle triangle,
    float& tClosest, float bClosest[3])
{
    Line line = <convert ray to line>;
    float sqrDistance = SquaredDistance(line, triangle, tClosest, bClosest);
    if (tClosest < 0)
    {
        sqrDistance = SquaredDistance(ray.P, triangle, bClosest);
        tClosest = 0;
    }
    return sqrDistance;
}
```

An advantage to sharing the line-triangle code in this manner is that if any numerical issues arise due to round-off errors or due to the choice of the epsilon threshold, they are all confined to a single function. This minimizes your maintenance of the source code—you do not have to fix the numerical problems in multiple functions. Another advantage of the encapsulation is that any performance improvements you can make to the line-triangle function will automatically lead to performance improvements in the ray-triangle function.

14.4.3 SEGMENT TO TRIANGLE

The segment-triangle distance function also makes use of the line-triangle distance function, just as the ray-triangle distance function did. The pseudocode is

```

float SquaredDistance (Segment segment, Triangle triangle,
    float& tClosest, float bClosest[3])
{
    Line line = <convert segment to line>;
    float sqrDistance = SquaredDistance(line, triangle, tClosest, bClosest);
    if (tClosest < segment.t0)
    {
        sqrDistance = SquaredDistance(segment.end0, triangle, bClosest);
        tClosest = t0;
    }
    else if (tClosest > segment.t1)
    {
        sqrDistance = SquaredDistance(segment.end1, triangle, bClosest);
        tClosest = t1;
    }
    return sqrDistance;
}

```

This implementation has the same advantages as mentioned for the ray-triangle distance code.

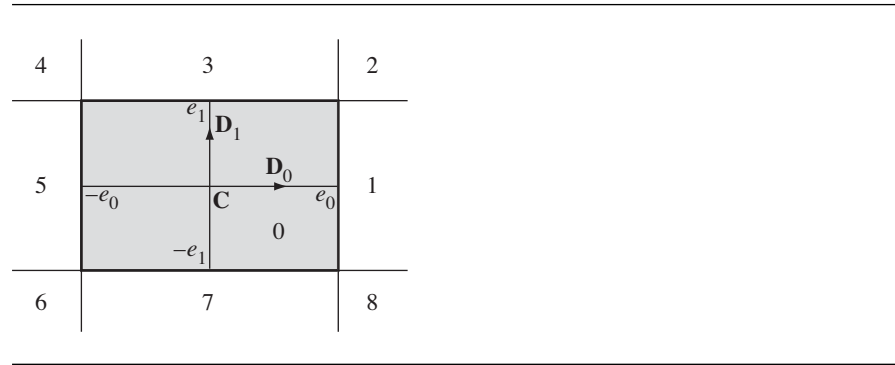
14.5 POINT TO RECTANGLE

The distance algorithm for point to rectangle is nearly the same as the distance algorithm for point to triangle. The rectangle is parameterized by $\mathbf{X}(s, t) = \mathbf{C} + s\mathbf{D}_0 + t\mathbf{D}_1$ for $|s| \leq e_0$ and $|t| \leq e_1$. The squared-distance function is the quadratic function

$$Q(s, t) = |\mathbf{X}(s, t) - \mathbf{P}|^2 = s^2 + t^2 + 2b_0s + 2b_1t + c$$

where $b_i = \mathbf{D}_i \cdot (\mathbf{C} - \mathbf{P})$ and $c = |\mathbf{C} - \mathbf{P}|^2$. The parameter plane is partitioned into nine regions by the lines $s = -e_0$, $s = e_0$, $t = -e_1$, and $t = e_1$. This partition is shown in Figure 14.3.

There is, however, one main difference. If the triangle's gradient of Q is zero in regions 2, 4, or 6 of Figure 14.1, then the minimum of Q can occur on one of two edges. For rectangles, this is not the case. If the rectangle's gradient of Q is zero in region 2, then the minimum must occur at the vertex. The same argument is made for regions 4, 6, and 8. Because the edges of the rectangle meet at a right angle, the level sets of the squared-distance function are in fact circles, not ellipses as was the case for the triangle's Q . The closest point on the rectangle to the specified point $\mathbf{P}$ is obtained by projecting $\mathbf{P}$ onto the plane of the rectangle; call this point $\mathbf{P}_0$. If $\mathbf{P}_0$ is inside the rectangle, then it is the closest point. If it is in regions 1, 3, 5, or 7, then the closest point is obtained by projecting $\mathbf{P}$ onto the rectangle edge for that region.

Figure 14.3 Partitioning of the st -plane by the rectangle domain.

Otherwise, $\mathbf{P}_0$ is in one of regions 2, 4, 6, or 8, and the closest point is the rectangle vertex of that region.

The projection of $\mathbf{P}$ onto the plane of the rectangle is $\mathbf{P}_0 = \mathbf{P} + s\mathbf{D}_0 + t\mathbf{D}_1$, where $s = \mathbf{D}_0 \cdot (\mathbf{P} - \mathbf{C})$ and $t = \mathbf{D}_1 \cdot (\mathbf{P} - \mathbf{C})$. Determination of the region containing the projection and the closest point to the projection requires a simple analysis of s and t . The pseudocode is

```
float SquaredDistance (Vector3 P, Rectangle rectangle,
    float rClosest[2])
{
    Vector3 diff = rectangle.C - P;
    float b0 = Dot(rectangle.D0,diff);
    float b1 = Dot(rectangle.D1,diff);
    float s = -b0, t = -b1;

    sqrDistance = diff.SquaredLength();

    if (s < -rectangle.e0)
    {
        s = -rectangle.e0;
    }
    else if (s > rectangle.e0)
    {
        s = rectangle.e0;
    }
    sqrDistance += s * (s + 2 * b0);

    if (t < -rectangle.e1)
```



```

{
    t = -rectangle.e1;
}
else if (t > rectangle.e1)
{
    t = rectangle.e1;
}
sqrDistance += t * (t + 2 * b1);

rClosest[0] = s;
rClosest[1] = t;
return sqrDistance;
}

```

14.6 LINEAR COMPONENT TO RECTANGLE

The computation of distance from a linear component to a rectangle is very similar to that of distance from a linear component to a triangle.

14.6.1 LINE TO RECTANGLE

An implementation must distinguish between the cases of parallel or nonparallel line and rectangle. Let the rectangle have center $\mathbf{C}$, directions $\mathbf{D}_0$ and $\mathbf{D}_1$, and extents e_0 and e_1 . A unit-length normal vector for the plane of the rectangle is $\mathbf{N} = \mathbf{D}_0 \times \mathbf{D}_1$. Let the line be $\mathbf{P} + t\mathbf{D}$. If $\mathbf{N} \cdot \mathbf{D} = 0$, the line and the rectangle are parallel. When using floating-point arithmetic, you will want to threshold the value, say, $|\mathbf{N} \cdot \mathbf{D}| \leq \varepsilon$ for a small tolerance ε .

If the line and rectangle are not parallel, the intersection point of the line and the plane of the rectangle is a solution to

$$\mathbf{P} + t\mathbf{D} = \mathbf{C} + s_0\mathbf{D}_0 + s_1\mathbf{D}_1$$

If $|s_0| \leq e_0$ and $|s_1| \leq e_1$, then the intersection point must be inside the rectangle, and the distance from the line to the rectangle is zero. If the intersection point is outside the rectangle, it is sufficient to compute the distance between the line and each edge of the rectangle, selecting the minimum of the four values to be the distance between the line and the rectangle.

Let $\mathbf{U}$ and $\mathbf{V}$ be vectors so that $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{D}$ are mutually perpendicular. Subtracting $\mathbf{C}$ from both sides of the equation defining the intersection:

$$s_0\mathbf{D}_0 + s_1\mathbf{D}_1 = (\mathbf{P} - \mathbf{C}) + t\mathbf{D}$$

Dot the equation with $\mathbf{U}$ and with $\mathbf{V}$ to obtain two linear equations in the two unknowns s_0 and s_1 . The solution is

$$\begin{bmatrix} s_0 \\ s_1 \end{bmatrix} = \begin{bmatrix} \mathbf{U} \cdot \mathbf{D}_0 & \mathbf{U} \cdot \mathbf{D}_1 \\ \mathbf{V} \cdot \mathbf{D}_0 & \mathbf{V} \cdot \mathbf{D}_1 \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{U} \cdot (\mathbf{P} - \mathbf{C}) \\ \mathbf{V} \cdot (\mathbf{P} - \mathbf{C}) \end{bmatrix}$$

Also dot the equation with $\mathbf{N}$ and solve for

$$t = \frac{-\mathbf{N} \cdot (\mathbf{P} - \mathbf{C})}{\mathbf{N} \cdot \mathbf{D}}$$

To avoid the division, dot the equation with $\mathbf{D}$ and solve for

$$t = s_0 \mathbf{D} \cdot \mathbf{D}_0 + s_1 \mathbf{D} \cdot \mathbf{D}_1 - \mathbf{D} \cdot (\mathbf{P} - \mathbf{C})$$

The pseudocode for the algorithm is listed next, returning the squared distance, the t value for the closest line point, and the s_0 and s_1 values for the closest rectangle point.

```
float SquaredDistance (Line line, Rectangle rectangle,
    float& tClosest, float sClosest[2])
{
    const float epsilon = <small tolerance, say, 1e-06>;
    Vector3 N = Cross(rectangle.D[0], rectangle.D[1]);
    float NdD = Dot(N, line.D);
    if (|NdD| > epsilon)
    {
        // The line and rectangle are not parallel. Compute
        // the intersection point.
        Vector3 PmC = line.P - rectangle.C;
        float UdD0 = Dot(line.U, rectangle.D[0]);
        float UdD1 = Dot(line.U, rectangle.D[1]);
        float UdPmC = Dot(line.U, PmC);
        float VdE0 = Dot(line.V, rectangle.D[0]);
        float VdE1 = Dot(line.V, rectangle.D[1]);
        float VdPmC = Dot(line.V, PmC);
        float invDet = 1/(UdD0 * VdD1 - UdD1 * VdD0);
        float s0 = (VdD1 * UdPmC - UdD1 * VdPmC) * invDet;
        float s1 = (UdD0 * VdPmC - VdD0 * UdPmC) * invDet;
        if (|s0| <= rectangle.e[0] && |s1| <= rectangle.e[1])
        {
            // The intersection point is inside the rectangle.
            tClosest = -Dot(N, PmC)/NdD;
            sClosest[0] = s0;
            sClosest[1] = s1;
        }
    }
}
```

```

        return 0;
    }
}

// The line and rectangle are not parallel and the line
// does not intersect the rectangle. Or the line and
// triangle are rectangle. The order of visitation is
// left edge, right edge, bottom edge, top edge.
float sqrDistance = infinity;
Segment seg;
for (i1 = 0; i1 < 2; i1++)
{
    for (i0 = 0; i0 < 2; i0++)
    {
        seg.P = rectangle.C +
            (2 * i0 - 1) * rectangle.e[i1] * rectangle.D[i1];
        seg.D = rectangle.D[1-i1];
        seg.e = rectangle.e[1-i1];
        float tmpLT, tmpST;
        float tmpSD = SquaredDistance(line, seg, tmpLT, tmpST);
        if (tmpSD < sqrDistance)
        {
            sqrDistance = tmpSD;
            tClosest = tmpT;
            float sb1 = -tmpST/seg.e;
            sClosest[0] = rectangle.e[0] * ((1 - i1) * (2 * i0 - 1)
                + i1 * sb1);
            sClosest[1] = rectangle.e[1] * ((1 - i0) * (2 * i1 - 1)
                + i0 * sb1);
        }
    }
}
return sqrDistance;
}

```

Just as **U** and **V** may be precomputed and stored with the line's data structure if the line will be used in multiple distance queries, you may precompute the normal vector **N** and store it with the rectangle's data structure if the rectangle will be used in multiple distance queries.

14.6.2 RAY TO RECTANGLE

This algorithm reuses the line-rectangle distance function. The line containing the ray is used in the line-rectangle distance function call. One of the return values is t ,

the parameter corresponding to the line point closest to the rectangle. If $t \geq 0$, then this point is on the ray and it is the closest ray point to the rectangle. However, if $t < 0$, then it must be the case that the ray origin is closest to the rectangle, in which case you must call a function to compute the distance between a point and a rectangle. The pseudocode is

```
float SquaredDistance (Ray ray, Rectangle rectangle,
    float& tClosest, float sClosest[2])
{
    Line line = <convert ray to line>;
    float sqrDistance = SquaredDistance(line, rectangle, tClosest, sClosest);
    if (tClosest < 0)
    {
        sqrDistance = SquaredDistance(ray.P, rectangle, sClosest);
        tClosest = 0;
    }
    return sqrDistance;
}
```

An advantage to sharing the line-rectangle code in this manner is that if any numerical issues arise due to round-off errors or due to the choice of the epsilon threshold, they are all confined to a single function. This minimizes your maintenance of the source code—you do not have to fix the numerical problems in multiple functions. Another advantage of the encapsulation is that any performance improvements you can make to the line-rectangle function will automatically lead to performance improvements in the ray-rectangle function.

14.6.3 SEGMENT TO RECTANGLE

The segment-rectangle distance function also makes use of the line-rectangle distance function, just as the ray-rectangle distance function did. The line containing the segment is used in the line-rectangle distance function call. To allow for your favorite choice of segment representation, suppose that the segment is $\mathbf{P} + t\mathbf{D}$ with $t \in [t_0, t_1]$. The return value t from the line-rectangle function call is compared to the segment t -interval. If $t \in [t_0, t_1]$, then the point is on the segment and it is the closest segment point to the rectangle. However, if $t \notin [t_0, t_1]$, then it must be the case that one of the segment endpoints is closest to the rectangle, and thus you must call a function to compute the distance between a point and a rectangle. The pseudocode is

```
float SquaredDistance (Segment segment, Rectangle rectangle,
    float& tClosest, float sClosest[2])
{
    Line line = <convert segment to line>;
    float sqrDistance = SquaredDistance(line, rectangle, tClosest, sClosest);
```

```

if (tClosest < segment.t0)
{
    sqrDistance = SquaredDistance(segment.end0, rectangle, sClosest);
    tClosest = t0;
}
else if (tClosest > segment.t1)
{
    sqrDistance = SquaredDistance(segment.end1, rectangle, sClosest);
    tClosest = t1;
}
}

```

This implementation has the same advantages as mentioned for the ray-rectangle distance code.

14.7 TRIANGLE OR RECTANGLE TO TRIANGLE OR RECTANGLE

Although it is possible to solve the constrained quadratic minimization for two objects, each of which is a triangle or a rectangle, an implementation is very tedious because the quadratic function depends on four parameters, two per object. The domain of the function is the 4D space. This space is partitioned into 49 regions by two triangles, 63 regions by a triangle and a rectangle, and 81 regions by two rectangles. The number of conditional blocks in an implementation is equal to the number of regions in the partition.

A simpler implementation is based on the observation that one of the two closest points on two objects can be chosen to be an edge point. The algorithm to compute the distance between objects computes the distances between the edges of one object to the other object, and vice versa. The minimum of these distances is the object-object distance. The pseudocode for triangle-triangle distance is listed next.

```

float SquaredDistance (Triangle tri0, Triangle tri1,
    float bClosest0[3], float bClosest1[3])
{
    // Compare edges of triangle0 to the interior of triangle1.
    float sqrDistance = infinity;
    float tmpSD, tmpT, tmpB[3];
    Segment seg;
    int i0, i1;
    for (i0 = 2, i1 = 0; i1 < 3; i0 = i1++)
    {
        seg.P = (tri0.P[i0] + tri0.P[i1])/2;
        seg.D = tri0.P[i1] - tri0.P[i0];
        seg.e = Normalize(seg.D)/2;
    }
}

```

```

    tmpSD = SquaredDistance(seg,tri1,tmpT,tmpB);
    if (tmpSD < sqrDistance)
    {
        sqrDistance = tmpSD;
        bClosest0[i0] = (1 - tmpT/seg.e)/2;
        bClosest0[i1] = 1 - bClosest0[i0];
        bClosest0[3 - i0 - i1] = 0;
        bClosest1[0] = tmpB[0];
        bClosest1[1] = tmpB[1];
        bClosest1[2] = tmpB[2];
    }
    if (sqrDistance is effectively zero)
    {
        return sqrDistance;
    }
}

// Compare edges of triangle1 to the interior of triangle0.
for (i0 = 2, i1 = 0; i1 < 3; i0 = i1++)
{
    seg.P = (tri1.P[i0] + tri1.P[i1])/2;
    seg.D = tri1.P[i1] - tri1.P[i0];
    seg.e = Normalize(seg.D)/2;
    tmpSD = SquaredDistance(seg,tri0,tmpT,tmpB);
    if (tmpSD < fSqrDist)
    {
        fSqrDist = tmpSD;
        bClosest1[i0] = (1 - tmpT/seg.e)/2;
        bClosest1[i1] = 1 - bClosest1[i0];
        bClosest1[3 - i0 - i1] = 0;
        bClosest0[0] = tmpB[0];
        bClosest0[1] = tmpB[1];
        bClosest0[2] = tmpB[2];
    }
    if (sqrDistance is effectively zero)
    {
        return sqrDistance;
    }
}

return fSqrDist;
}

```

The `Normalize` function makes the vector unit length and returns the length of the original vector.

The triangle-rectangle and rectangle-rectangle squared-distance functions are similarly implemented. The array of closest values for the rectangle are computed using the same expressions shown in the line-rectangle pseudocode.

14.8 POINT TO ORIENTED BOX

The squared-distance algorithm treats the box as a solid. Any point inside the box has distance zero from the box. Let the box have center $\mathbf{C}$, orthonormal axes $\mathbf{U}_i$, and extents e_i for $0 \leq i \leq 2$. Let the point be written as $\mathbf{P} = \mathbf{C} + s_0\mathbf{U}_0 + s_1\mathbf{U}_1 + s_2\mathbf{U}_2$. Solving for the coefficients yields $s_i = \mathbf{U}_i \cdot (\mathbf{P} - \mathbf{C})$ for all i . Depending on the values of (s_0, s_1, s_2) relative to the parameter domain $[-e_0, e_0] \times [-e_1, e_1] \times [-e_2, e_2]$, the closest point is either $\mathbf{P}$ itself, a face point, an edge point, or a vertex. The pseudocode is

```
float SquaredDistance (Vector3 P, Box box, float bClosest[3])
{
    Vector3 diff = P - box.C;
    float sqrDistance = 0, delta;
    for (i = 0; i < 3; i++)
    {
        bClosest[i] = Dot(box.axis[i], diff);
        if (bClosest[i] < -box.e[i])
        {
            delta = bClosest[i] + box.e[i];
            sqrDistance += delta * delta;
            bClosest[i] = -box.e[i];
        }
        else if (bClosest[i] > box.e[i])
        {
            delta = bClosest[i] - box.e[i];
            sqrDistance += delta * delta;
            bClosest[i] = box.e[i];
        }
    }
    return sqrDistance;
}
```

14.9 LINEAR COMPONENT TO ORIENTED BOX

The distance algorithm may be set up as a constrained quadratic minimization. The line has one parameter and the box has three parameters, so the squared-distance function has four parameters.

14.9.1 **LINE TO ORIENTED BOX**

A simple method for computing the distance between a line and an oriented box is to iterate over the six faces of the box and compute the distance from the line to those faces. If the line intersects a face, the distance is zero. The pseudocode is

```
float SquaredDistance (Line line, Box box, float& lClosest,
    float bClosest[3])
{
    float sqrDistance = infinity;
    float tmpSD, tmpLC, tmpFC[2];
    Rectangle face;

    // Process the box face at -e0.
    face.C = box.C - box.e[0] * box.axis[0];
    face.axis[0] = box.axis[1];
    face.axis[1] = box.axis[2];
    face.e[0] = box.e[1];
    face.e[1] = box.e[2];
    tmpSD = SquaredDistance(line, face, tmpLC, tmpFC);
    if (tmpSD < sqrDistance)
    {
        sqrDistance = tmpSD;
        lClosest = tmpLC;
        bClosest[0] = -box.e[0];
        bClosest[1] = tmpFC[0];
        bClosest[2] = tmpFC[1];
        if (sqrDistance is effectively zero)
        {
            return sqrDistance;
        }
    }

    // Process the box face at +e0.
    // Process the box face at -e1.
    // Process the box face at +e1.
    // Process the box face at -e2.
    // Process the box face at +e2.

    return sqrDistance;
}
```

EXERCISE 14.1 Implement the five unfinished cases in the pseudocode for computing the squared distance between a line and an oriented box. ■

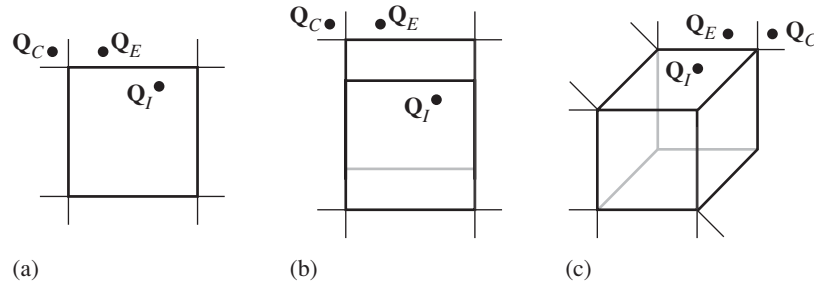


Figure 14.4 Projections of the oriented box onto a plane perpendicular to $\mathbf{E}$. (a) $\mathbf{E} = (*, 0, 0)$ has two zeros. (b) $\mathbf{E} = (0, *, *)$ has one zero. (c) $\mathbf{E}$ has no zeros. In all cases, the line projects to a point.

The iteration over the faces is relatively inefficient, doing more work than is necessary to find a closest point. An alternative is provided here, but involves a more sophisticated implementation.

The line is parameterized by $\mathbf{P} + t\mathbf{D}$ for any real number t . The box is parameterized by $\mathbf{C} + s_0\mathbf{D}_0 + s_1\mathbf{D}_1 + s_2\mathbf{D}_2$ for $|s_i| \leq e_i$ for all i . The line is first transformed to the coordinate system of the box, say, $\mathbf{Q} + t\mathbf{E}$, where

$$\mathbf{Q} = (\mathbf{D}_0 \cdot (\mathbf{P} - \mathbf{C}), \mathbf{D}_1 \cdot (\mathbf{P} - \mathbf{C}), \mathbf{D}_2 \cdot (\mathbf{P} - \mathbf{C})), \quad \mathbf{E} = (\mathbf{D}_0 \cdot \mathbf{D}, \mathbf{D}_1 \cdot \mathbf{D}, \mathbf{D}_2 \cdot \mathbf{D})$$

The vector $\mathbf{E}$ is necessarily unit length. Three cases are of interest based on the number of zero components of $\mathbf{E}$. Figure 14.4 shows projections of the oriented box onto a plane perpendicular to $\mathbf{E}$.

In Figure 14.4 (a), the oriented box projects to a rectangle. The plane of the projection is spanned by $\mathbf{D}_1$ and $\mathbf{D}_2$. The other vector $\mathbf{D}_0$ is either out of, or into, the plane of the page. The projected box has one visible face. In Figure 14.4 (b), the oriented box also projects to a rectangle, but it has two visible faces. The vector $\mathbf{D}_0$ is either to the right or to the left on the page. In Figure 14.4 (c), the oriented box projects to a hexagon with three pairs of parallel edges. Three faces are visible. In all cases, the projections imply a partitioning of the plane into regions, which are called *Voronoi regions*. Each region has a feature of the box to which the region points are closest. The features are classified as the interior of the projection, an edge of the projection, or a vertex of the projection. In (a), (b), and (c) of the figure, three projected lines are shown.

A line that projects to a point $\mathbf{Q}_I$ is an interior point. The line intersects the oriented box, so the distance is zero. The implementation includes a point-in-convex-polygon query to determine if the projected line is of this type.

A line that projects to a point $\mathbf{Q}_E$ is closest to an edge of the projection. The projection point is outside the polygon. The implementation determines this with its

point-in-convex-polygon query. Additional logic is included to select which edge Q_E is closest to. The final step is to compute the (squared) distance from the line to that edge. Previously in this chapter we have covered the topic of line-segment distance calculations.

A line that projects to a point Q_C is closest to a corner of the projection. The projection point is outside the polygon. The implementation determines this with its point-in-convex-polygon query. Additional logic is included to select which corner Q_C is closest to. The final step is to compute the (squared) distance from the line to that corner. Previously in this chapter we have covered the topic of point-line distance calculations.

Although the illustrations in Figure 14.4 make it appear as if the implementation should be simple, the details are substantial. The CD-ROM does have an implementation of this algorithm, but the code is long enough that I do not want to waste pages with a printout.

14.9.2 RAY TO ORIENTED BOX

Once again we rely on reusing code so that numerical problems are restricted to small pieces of code and to minimize the amount of code maintenance. The line-box squared-distance function is called and the return line parameter is clamped to $[0, \infty)$. The pseudocode for the ray-box squared-distance function is

```
float SquaredDistance (Ray ray, Box box, float& rClosest,
    float bClosest[3])
{
    Line line = <convert ray to line>;
    float sqrDistance = SquaredDistance(line, box, rClosest, bClosest);
    if (rClosest < 0)
    {
        rClosest = 0;
        sqrDistance = SquaredDistance(ray.P, box, bClosest);
    }
    return sqrDistance;
}
```

14.9.3 SEGMENT TO ORIENTED BOX

And yet again we rely on reusing code so that numerical problems are restricted to small pieces of code and to minimize the amount of code maintenance. The line-box squared-distance function is called and the return line parameter is clamped to $[-e, e]$. The pseudocode for the segment-box squared-distance function is

```

float SquaredDistance (Segment segment, Box box, float& sClosest,
                      float bClosest[3])
{
    Line line = <convert segment to line>;
    float sqrDistance = SquaredDistance(line, box, sClosest, bClosest);
    Vector3 end;
    if (sClosest < -segment.e)
    {
        sClosest = -segment.e;
        end = segment.P - segment.e * segment.D;
        sqrDistance = SquaredDistance(end, box, bClosest);
    }
    else if (sClosest > segment.e)
    {
        sClosest = segment.e;
        end = segment.P + segment.e * segment.D;
        sqrDistance = SquaredDistance(end, box, bClosest);
    }
    return sqrDistance;
}

```

14.10 TRIANGLE TO ORIENTED BOX

As a constrained quadratic minimization, the squared-distance function has five parameters, two for the triangle and three for the box. The number of cases involved in the partitioning 5D space by the domains of the triangle parameters and box parameters is enormous. A simpler approach is to compute distances from the triangle to the box faces, choosing the smallest distance as the triangle-box distance. However, it is possible that this distance is positive, but the triangle is contained in the box. To trap this case, we check if any vertex is inside the box. If it is, then the distance is zero. Now it is possible that all vertices are outside the box, but a portion of the triangle is inside the box. The distance is zero in this case, but the triangle-face tests will determine this. The pseudocode is

```

float SquaredDistance (Triangle triangle, Box box,
                      float tClosest[2], float bClosest[3])
{
    for (i = 0; i < 3; i++)
    {
        if (PointInBox(triangle.P[i]))
        {
            tClosest[0] = i * (2 - i);
            tClosest[1] = i * (i - 1)/2;

```

```

        Vector3f diff = triangle.P[i] - box.C;
        for (j = 0; j < 3; j++)
        {
            bClosest[j] = Dot(box.axis[j],diff);
        }
        return 0;
    }
}

float sqrDistance = infinity;
float tmpSD, tmpTC[2], tmpFC[2];
Rectangle face;

// Process the box face at -e0.
face.C = box.C - box.e[0] * box.axis[0];
face.axis[0] = box.axis[1];
face.axis[1] = box.axis[2];
face.e[0] = box.e[1];
face.e[1] = box.e[2];
tmpSD = SquaredDistance(triangle,face,tmpTC,tmpFC);
if (tmpSD < sqrDistance)
{
    sqrDistance = tmpSD;
    tClosest[0] = tmpTC[0];
    tClosest[1] = tmpTC[1];
    bClosest[0] = -box.e[0];
    bClosest[1] = tmpFC[0];
    bClosest[2] = tmpFC[1];
    if (sqrDistance is effectively zero)
    {
        return sqrDistance;
    }
}

// Process the box face at +e0.
// Process the box face at -e1.
// Process the box face at +e1.
// Process the box face at -e2.
// Process the box face at +e2.

return sqrDistance;
}

```

EXERCISE 14.2 Implement the five unfinished cases in the pseudocode for computing the squared distance between a triangle and an oriented box. ■

14.11 RECTANGLE TO ORIENTED BOX

The algorithm for computing the squared distance between a rectangle and an oriented box may be chosen similarly to that for a triangle and an oriented box. The pseudocode is

```
float SquaredDistance (Rectangle rectangle, Box box,
    float rClosest[2], float bClosest[3])
{
    for (i1 = 0; i1 < 2; i1++)
    {
        float param1 = (2 * i1 - 1) * rectangle.e[1];
        Vector3 delta1 = param1 * rectangle.axis[1];
        for (i0 = 0; i0 < 2; i0++)
        {
            float param0 = (2 * i0 - 1) * rectangle.e[0];
            Vector3 delta0 = param0 * rectangle.axis[0];
            Vector3 corner = rectangle.C + delta0 + delta1;
            if (PointInBox(corner))
            {
                tClosest[0] = param0;
                tClosest[1] = param1;
                Vector3f diff = corner - box.C;
                for (j = 0; j < 3; j++)
                {
                    bClosest[j] = Dot(box.axis[j], diff);
                }
                return 0;
            }
        }
    }

    float sqrDistance = infinity;
    float tmpSD, tmpRC[2], tmpFC[2];
    Rectangle face;

    // Process the box face at -e0.
    face.C = box.C - box.e[0] * box.axis[0];
    face.axis[0] = box.axis[1];
    face.axis[1] = box.axis[2];
    face.e[0] = box.e[1];
    face.e[1] = box.e[2];
    tmpSD = SquaredDistance(rectangle, face, tmpRC, tmpFC);
    if (tmpSD < sqrDistance)
```

```

    {
        sqrDistance = tmpSD;
        tClosest[0] = tmpRC[0];
        tClosest[1] = tmpRC[1];
        bClosest[0] = -box.e[0];
        bClosest[1] = tmpFC[0];
        bClosest[2] = tmpFC[1];
        if (sqrDistance is effectively zero)
        {
            return sqrDistance;
        }
    }

    // Process the box face at +e0.
    // Process the box face at -e1.
    // Process the box face at +e1.
    // Process the box face at -e2.
    // Process the box face at +e2.

    return sqrDistance;
}

```

EXERCISE 14.3 Implement the five unfinished cases in the pseudocode for computing the squared distance between a rectangle and an oriented box. ■

14.12 ORIENTED BOX TO ORIENTED BOX

Quite a few options exist for computing the distance between two oriented boxes. The trade-off is mainly performance versus ease of implementation. The simplest algorithm to implement makes use of the rectangle-box distance queries. If any vertex of one box is contained in the other box, the distance between boxes is zero; otherwise, compute the distance from the six faces of one box to the other box and select the minimum value (which is zero if the boxes overlap). The pseudocode is

```

float SquaredDistance (Box box0, Box box1, float b0Closest[3],
                      float b1Closest[3])
{
    for each vertex P of box0 do
    {
        if PointInBox(P, box1)
        {
            set b0Closest values;
            set b1Closest values;
        }
    }
}

```

```

        return 0;
    }
}

float sqrDistance = infinity;
float tmpSD, tmpRC[2], tmpBC[3];
Rectangle face;
for each face of box0 do
{
    float tmpSD = SquaredDistance(face,box1,tmpRC,tmpBC);
    if (tmpSD < sqrDistance)
    {
        sqrDistance = tmpSD;
        set b0Closest from tmpRC values and a box0 extent;
        set b1Closest to tmpBC1 values;
        if (sqrDistance is effectively zero)
        {
            return sqrDistance;
        }
    }
}
return sqrDistance;
}

```

The point-in-box queries are relatively inexpensive, but the rectangle-box queries are not. Also, the point-in-box queries are not an exact test for overlap. If all the vertices of box0 are outside box1, it is possible that the two boxes overlap. To improve the performance, the point-in-box queries may be replaced by a query that uses the *method of separating axes*. See Section 8.1 for a discussion of this method. The separation query is easy enough to implement, but the complication is that the format of the SquaredDistance functions requires you to compute the box coordinates for a closest point. The method of separating axes does not immediately reveal such a point. The point-in-box queries are replaced by the pseudocode

```

float SquaredDistance (Box box0, Box box1, float b0Closest[3],
    float b1Closest[3])
{
    if Separated(box0,box1) then // easy to implement
    {
        set b0Closest values; // hard to implement
        set b1Closest values; // hard to implement
        return 0;
    }
    // ... remainder of previous pseudocode ...
}

```

Now if your applications do not need to know the values `b0Closest` and `b1Closest`, then the variation provided here should perform better on average than the previous pseudocode.

Yet another possibility, one that has very good performance but only in exchange for an implementation that is tedious to create and difficult to make robust, is the *GJK algorithm* [GJK88, GF90, Cam97, vdB99]. The details are significant and not something I want to focus on in this book. A book that does cover all the details and has a source code implementation is [vdB03]. The algorithm is also covered in some detail in [Eri04], which also has the best coverage of collision detection algorithms of any book I have read.

14.13 MISCELLANEOUS

A library of distance calculation methods can be arbitrarily complex. There are many other cases that can arise in an application. These cases might require distance calculations not specifically derived here: parallelogram to point, segment, rectangle, or parallelogram; and parallelepiped to point, segment, rectangle, parallelogram, or parallelepiped. Many of these may be directly implemented using the ideas discussed in this chapter. Other cases might involve distance from point to quadric surface; from point to circle (in 3D) or disk; from point to cylinder; from line segment to these same quadratic-style objects; ad infinitum. At any rate, such a library is never complete and will continually evolve.

14.13.1 POINT TO ELLIPSE

We only need to solve this problem when the ellipse is axis aligned. Oriented ellipses can be rotated and translated to an axis-aligned ellipse centered at the origin and the distance can be measured in that system. The basic idea can be found in an article by John Hart (on computing distance, but between point and ellipsoid) in [Hec94].

Let (u, v) be the point in question. Let the ellipse be $(x/a)^2 + (y/b)^2 = 1$. The closest point (x, y) on the ellipse to (u, v) must occur so that $(x - u, y - v)$ is normal to the ellipse. Since an ellipse normal is $\nabla((x/a)^2 + (y/b)^2) = (x/a^2, y/b^2)$, the orthogonality condition implies that $u - x = tx/a^2$ and $v - y = ty/b^2$ for some t . Solving yields $x = a^2u/(t + a^2)$ and $y = b^2v/(t + b^2)$. Replacing in the ellipse equation yields

$$\left(\frac{au}{t + a^2}\right)^2 + \left(\frac{bv}{t + b^2}\right)^2 = 1$$

Multiplying through by the denominators yields the quartic polynomial

$$F(t) = (t + a^2)^2(t + b^2)^2 - a^2u^2(t + b^2)^2 - b^2v^2(t + a^2)^2 = 0$$

The largest root $\bar{t}$ of the polynomial corresponds to the closest point on the ellipse.

A closed-form solution for the roots of a quartic polynomial exists and can be used to compute the largest root. This root also can be found by a Newton's iteration scheme. If (u, v) is inside the ellipse, then $t_0 = 0$ is a good initial guess for the iteration. If (u, v) is outside the ellipse, then $t_0 = \max\{a, b\}\sqrt{u^2 + v^2}$ is a good initial guess. The iteration itself is

$$t_{i+1} = t_i - F(t_i)/F'(t_i), \quad i \geq 0$$

Some numerical issues need to be addressed. For (u, v) near the coordinate axes, the algorithm is ill-conditioned because of the divisions of values near zero in the equations relating (x, y) to (u, v) . Those cases need to be handled separately. Also, if a and b are large, then $F(t_i)$ can be quite large. In these cases consider uniformly scaling the data to $O(1)$ as floating-point numbers first, computing the distance, then rescaling to get the distance in the original coordinates.

14.13.2 POINT TO ELLIPSOID

The method of measuring distance is a straightforward generalization of that for an ellipse. Let (u, v, w) be the point in question. Let the ellipsoid be $(x/a)^2 + (y/b)^2 + (z/c)^2 = 1$. The closest point (x, y, z) on the ellipsoid to (u, v, w) must occur so that $(x - u, y - v, z - w)$ is normal to the ellipsoid. Since an ellipsoid normal is $\nabla((x/a)^2 + (y/b)^2 + (z/c)^2) = (x/a^2, y/b^2, z/c^2)$, the orthogonality condition implies that $u - x = tx/a^2$, $v - y = ty/b^2$, and $w - z = tz/c^2$ for some t . Solving yields $x = a^2u/(t + a^2)$, $y = b^2v/(t + b^2)$, and $z = c^2w/(t + c^2)$. Replacing in the ellipsoid equation yields

$$\left(\frac{au}{t + a^2}\right)^2 + \left(\frac{bv}{t + b^2}\right)^2 + \left(\frac{cw}{t + c^2}\right)^2 = 1$$

Multiplying through by the denominators yields the sixth-degree polynomial

$$\begin{aligned} F(t) = & (t + a^2)^2(t + b^2)^2(t + c^2)^2 - a^2u^2(t + b^2)^2(t + c^2)^2 \\ & - b^2v^2(t + a^2)^2(t + c^2)^2 - c^2w^2(t + a^2)^2(t + b^2)^2 = 0 \end{aligned}$$

The largest root $\bar{t}$ of the polynomial corresponds to the closest point on the ellipsoid.

The largest root can be found by a Newton's iteration scheme. If (u, v, w) is inside the ellipsoid, then $t_0 = 0$ is a good initial guess for the iteration. If (u, v, w) is outside the ellipsoid, then $t_0 = \max\{a, b, c\}\sqrt{u^2 + v^2 + w^2}$ is a good initial guess. The iteration method is the same as before, $t_{i+1} = t_i - F(t_i)/F'(t_i)$ for $i \geq 0$. The same numerical issues that occur in the ellipse problem need to be addressed for ellipsoids. For (u, v, w) near the coordinate planes, the algorithm is ill-conditioned because of the divisions of values near zero in the equations relating (x, y, z) to (u, v, w) . These cases can be handled separately. Also, if a , b , and c are large, $F(t_i)$ can be

quite large. In these cases consider uniformly scaling the data to $O(1)$ as floating-point numbers first, computing the distance, then rescaling to get the distance in the original coordinates.

14.13.3 POINT TO QUADRATIC CURVE OR TO QUADRIC SURFACE

This subsection describes an algorithm for computing the distance from a point in 2D to a general quadratic curve defined implicitly by a second-degree quadratic equation in two variables, or from a point in 3D to a general quadric surface defined implicitly by a second-degree quadratic equation in three variables. I will use the term *object* to refer to either a curve (in 2D) or a surface (in 3D).

The general quadratic equation is

$$Q(\mathbf{X}) = \mathbf{X}^T \mathbf{A} \mathbf{X} + \mathbf{b}^T \mathbf{X} + c = 0$$

where A is a symmetric $N \times N$ matrix ($N = 2$ or $N = 3$, not necessarily invertible, for example, in the case of a cylinder or paraboloid), $\mathbf{b}$ is an $N \times 1$ vector, and c is a scalar. The parameter is $\mathbf{X}$, an $N \times 1$ vector. Given the surface $Q(\mathbf{X}) = 0$ and a point $\mathbf{Y}$, find the distance from $\mathbf{Y}$ to the object and compute a closest point $\mathbf{X}$.

Geometrically, the closest point $\mathbf{X}$ on the object to $\mathbf{Y}$ must satisfy the condition that $\mathbf{Y} - \mathbf{X}$ is normal to the object. Since the gradient $\nabla Q(\mathbf{X})$ is normal to the object, the algebraic condition for the closest point is

$$\mathbf{Y} - \mathbf{X} = t \nabla Q(\mathbf{X}) = t(2\mathbf{A}\mathbf{X} + \mathbf{b})$$

for some scalar t . Therefore,

$$\mathbf{X} = (I + 2tA)^{-1}(\mathbf{Y} - t\mathbf{b})$$

where I is the identity matrix. You could replace this equation for $\mathbf{X}$ into the general quadratic equation to obtain a polynomial in t of at most sixth degree.

Instead of immediately replacing $\mathbf{X}$ in the quadratic equation, the problem can be reduced to something simpler to code. Factor A using an eigendecomposition to obtain $A = RDR^T$, where R is an orthonormal matrix whose columns are eigenvectors of A and where D is a diagonal matrix whose diagonal entries are the eigenvalues of A . Then

$$\begin{aligned} \mathbf{X} &= (I + 2tA)^{-1}(\mathbf{Y} - t\mathbf{b}) \\ &= (RR^T + 2tRDR^T)^{-1}(\mathbf{Y} - t\mathbf{b}) \\ &= [R(I + 2tD)R^T]^{-1}(\mathbf{Y} - t\mathbf{b}) \\ &= R(I + 2tD)^{-1}R^T(\mathbf{Y} - t\mathbf{b}) \\ &= R(I + 2tD)^{-1}(\boldsymbol{\alpha} - t\boldsymbol{\beta}) \end{aligned}$$

where the last equation defines α and β . Replacing in the quadratic equation and simplifying yields

$$0 = (\alpha - t\beta)^T(I + 2tD)^{-1}D(I + 2tD)^{-1}(\alpha - t\beta) + \beta^T(I + 2tD)^{-1}(\alpha - t\beta) + c$$

The inverse diagonal matrix is

$$(I + 2tD)^{-1} = \text{Diag}\{1/(1 + 2td_0), 1/(1 + 2td_1)\}$$

for 2D or

$$(I + 2tD)^{-1} = \text{Diag}\{1/(1 + 2td_0), 1/(1 + 2td_1), 1/(1 + 2td_2)\}$$

for 3D. Multiplying through by $((1 + 2td_0)(1 + 2td_1))^2$ in 2D leads to a polynomial of at most fourth degree. Multiplying through by $((1 + 2td_0)(1 + 2td_1)(1 + 2td_2))^2$ in 3D leads to a polynomial equation of at most sixth degree.

The roots of the polynomial are computed and $\mathbf{X} = (I + 2tA)^{-1}(\mathbf{Y} - t\mathbf{b})$ is computed for each root t . The distances between $\mathbf{X}$ and $\mathbf{Y}$ are computed and the minimum distance is selected from them.

14.13.4 POINT TO CIRCLE IN 3D

A circle in 3D is represented by a center $\mathbf{C}$, a radius R , and a plane containing the circle, $\mathbf{N} \cdot (\mathbf{X} - \mathbf{C}) = 0$, where $\mathbf{N}$ is a unit-length normal to the plane. If $\mathbf{U}$ and $\mathbf{V}$ are also unit-length vectors so that $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{N}$ form a right-handed, orthonormal set, then the circle is parameterized as

$$\mathbf{X} = \mathbf{C} + R(\cos(\theta)\mathbf{U} + \sin(\theta)\mathbf{V}) = \mathbf{C} + R\mathbf{W}(\theta)$$

for angles $\theta \in [0, 2\pi)$. The last equation defines the vector $\mathbf{W}(\theta)$. Note that $|\mathbf{X} - \mathbf{C}| = R$, so the $\mathbf{X}$ -values are all equidistant from $\mathbf{C}$. Moreover, $\mathbf{N} \cdot (\mathbf{X} - \mathbf{C}) = 0$ since $\mathbf{U}$ and $\mathbf{V}$ are perpendicular to $\mathbf{N}$, so the $\mathbf{X}$ -values lie in the plane.

For each angle $\theta \in [0, 2\pi)$, the squared distance from a specified point $\mathbf{P}$ to the corresponding circle point is

$$F(\theta) = |\mathbf{C} + R\mathbf{W}(\theta) - \mathbf{P}|^2 = R^2 + |\mathbf{C} - \mathbf{P}|^2 + 2R(\mathbf{C} - \mathbf{P}) \cdot \mathbf{W}$$

The problem is to minimize $F(\theta)$ by finding θ_0 such that $F(\theta_0) \leq F(\theta)$ for all $\theta \in [0, 2\pi)$. Since F is a periodic and differentiable function, the minimum must occur when $F'(\theta) = 0$. Also, note that $(\mathbf{C} - \mathbf{P}) \cdot \mathbf{W}$ should be negative and as large in magnitude as possible to reduce the right-hand side in the definition of F . The derivative is

$$F'(\theta) = 2R(\mathbf{C} - \mathbf{P}) \cdot \mathbf{W}'(\theta)$$

where $\mathbf{W} \cdot \mathbf{W}' = 0$ since $\mathbf{W} \cdot \mathbf{W} = 1$ for all θ . The vector $\mathbf{W}'$ is unit length since $\mathbf{W}'' = -\mathbf{W}$ and $0 = \mathbf{W} \cdot \mathbf{W}'$ implies

$$0 = \mathbf{W} \cdot \mathbf{W}'' + \mathbf{W}' \cdot \mathbf{W}' = -1 + \mathbf{W}' \cdot \mathbf{W}'$$

Finally, $\mathbf{W}'$ is perpendicular to $\mathbf{N}$ since $\mathbf{N} \cdot \mathbf{W} = 0$ implies $0 = \mathbf{N} \cdot \mathbf{W}'$. All conditions imply that $\mathbf{W}$ is parallel to the projection of $\mathbf{P} - \mathbf{C}$ onto the plane and points in the same direction.

Let $\mathbf{Q}$ be the projection of $\mathbf{P}$ onto the plane. Then

$$\mathbf{Q} - \mathbf{C} = \mathbf{P} - \mathbf{C} - (\mathbf{N} \cdot (\mathbf{P} - \mathbf{C})) \mathbf{N}$$

The vector $\mathbf{W}(\theta)$ must be the normalized projection $(\mathbf{Q} - \mathbf{C})/|\mathbf{Q} - \mathbf{C}|$. The closest point on the circle to $\mathbf{P}$ is

$$\mathbf{X} = \mathbf{C} + R \frac{\mathbf{Q} - \mathbf{C}}{|\mathbf{Q} - \mathbf{C}|}$$

assuming that $\mathbf{Q} \neq \mathbf{C}$. The distance from point to circle is then $|\mathbf{P} - \mathbf{X}|$.

If the projection of $\mathbf{P}$ is exactly the circle center $\mathbf{C}$, then all points on the circle are equidistant from $\mathbf{C}$. The distance from point to circle is the length of the hypotenuse of any right triangle whose vertices are $\mathbf{C}$, $\mathbf{P}$, and any circle point. The lengths of the adjacent and opposite triangle sides are R and $|\mathbf{P} - \mathbf{C}|$, so the distance from point to circle is $\sqrt{R^2 + |\mathbf{P} - \mathbf{C}|^2}$.

14.13.5 CIRCLE TO CIRCLE IN 3D

The previous subsection described the formulation for a circle in three dimensions. Using this formulation, let the two circles be $\mathbf{C}_0 + R_0 \mathbf{W}_0(\theta)$ for $\theta \in [0, 2\pi)$ and $\mathbf{C}_1 + R_1 \mathbf{W}_1(\phi)$ for $\phi \in [0, 2\pi)$. The squared distance between any two points on the circles is

$$\begin{aligned} F(\theta, \phi) &= |\mathbf{C}_1 - \mathbf{C}_0 + R_1 \mathbf{W}_1 - R_0 \mathbf{W}_0|^2 \\ &= |\mathbf{D}|^2 + R_0^2 + R_1^2 + 2R_1 \mathbf{D} \cdot \mathbf{W}_1 - 2R_0 R_1 \mathbf{W}_0 \cdot \mathbf{W}_1 - 2R_0 \mathbf{D} \cdot \mathbf{W}_0 \end{aligned}$$

where $\mathbf{D} = \mathbf{C}_1 - \mathbf{C}_0$. Since F is doubly periodic and continuously differentiable, its global minimum must occur when $\nabla F = (0, 0)$. The partial derivatives are

$$\frac{\partial F}{\partial \theta} = -2R_0 \mathbf{D} \cdot \mathbf{W}'_0 - 2R_0 R_1 \mathbf{W}'_0 \cdot \mathbf{W}_1$$

and

$$\frac{\partial F}{\partial \phi} = 2R_1 \mathbf{D} \cdot \mathbf{W}'_1 - 2R_0 R_1 \mathbf{W}_0 \cdot \mathbf{W}'_1$$

Define $c_0 = \cos(\theta)$, $s_0 = \sin(\theta)$, $c_1 = \cos(\phi)$, and $s_1 = \sin(\phi)$. Then $\mathbf{W}_0 = c_0\mathbf{U}_0 + s_0\mathbf{V}_0$, $\mathbf{W}_1 = c_1\mathbf{U}_1 + s_1\mathbf{V}_1$, $\mathbf{W}'_0 = -s_0\mathbf{U}_0 + c_0\mathbf{V}_0$, and $\mathbf{W}'_1 = -s_1\mathbf{U}_1 + c_1\mathbf{V}_1$. Setting the partial derivatives equal to zero leads to

$$0 = s_0(a_0 + a_1c_1 + a_2s_1) + c_0(a_3 + a_4c_1 + a_5s_1)$$

$$0 = s_1(b_0 + b_1c_0 + b_2s_0) + c_1(b_3 + b_4c_0 + b_5s_0)$$

where

$$a_0 = -\mathbf{D} \cdot \mathbf{U}_0 \quad b_0 = -\mathbf{D} \cdot \mathbf{U}_1$$

$$a_1 = -R_1\mathbf{U}_0 \cdot \mathbf{U}_1 \quad b_1 = R_0\mathbf{U}_0 \cdot \mathbf{U}_1$$

$$a_2 = -R_1\mathbf{U}_0 \cdot \mathbf{V}_1 \quad b_2 = R_0\mathbf{U}_1 \cdot \mathbf{V}_0$$

$$a_3 = \mathbf{D} \cdot \mathbf{V}_0 \quad b_3 = \mathbf{D} \cdot \mathbf{V}_1$$

$$a_4 = R_1\mathbf{U}_1 \cdot \mathbf{V}_0 \quad b_4 = -R_0\mathbf{U}_0 \cdot \mathbf{V}_1$$

$$a_5 = R_1\mathbf{V}_0 \cdot \mathbf{V}_1 \quad b_5 = -R_0\mathbf{V}_0 \cdot \mathbf{V}_1$$

In matrix form,

$$\begin{aligned} \begin{bmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{bmatrix} \begin{bmatrix} s_0 \\ c_0 \end{bmatrix} &= \begin{bmatrix} a_0 + a_1c_1 + a_2s_1 & a_3 + a_4c_1 + a_5s_1 \\ b_2s_1 + b_5c_1 & b_1s_1 + b_4c_1 \end{bmatrix} \begin{bmatrix} s_0 \\ c_0 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ -(b_0s_1 + b_3c_1) \end{bmatrix} = \begin{bmatrix} 0 \\ \lambda \end{bmatrix} \end{aligned}$$

Let M denote the 2×2 matrix on the left-hand side of the equation. Multiplying by the adjoint of M yields

$$\det(M) \begin{bmatrix} s_0 \\ c_0 \end{bmatrix} = \begin{bmatrix} m_{11} & -m_{01} \\ -m_{10} & m_{00} \end{bmatrix} \begin{bmatrix} 0 \\ \lambda \end{bmatrix} = \begin{bmatrix} -m_{01}\lambda \\ m_{00}\lambda \end{bmatrix} \quad (14.4)$$

Summing the squares of the vector components and using $s_0^2 + c_0^2 = 1$ yields

$$(m_{00}m_{11} - m_{01}m_{10})^2 = \lambda^2 (m_{00}^2 + m_{01}^2)$$

The preceding equation can be reduced to a polynomial of degree 8 whose roots $c_1 \in [-1, 1]$ are the candidates to provide the global minimum of F . Formally computing the determinant and using $s_1^2 = 1 - c_1^2$ leads to

$$m_{00}m_{11} - m_{01}m_{10} = p_0(c_1) + s_1p_1(c_1)$$

where $p_0(z) = \sum_{i=0}^2 p_{0i}z^i$ and $p_1(z) = \sum_{i=0}^1 p_{1i}z$. The coefficients are

$$p_{00} = a_2b_1 - a_5b_2$$

$$p_{01} = a_0b_4 - a_3b_5$$

$$p_{02} = a_5b_2 - a_2b_1 + a_1b_4 - a_4b_5$$

$$p_{10} = a_0b_1 - a_3b_2$$

$$p_{11} = a_1b_1 - a_5b_5 + a_2b_4 - a_4b_2$$

Similarly,

$$m_{00}^2 + m_{01}^2 = q_0(c_1) + s_1q_1(c_1)$$

where $q_0(z) = \sum_{i=0}^2 q_{0i}z^i$ and $q_1(z) = \sum_{i=0}^1 q_{1i}z$. The coefficients are

$$q_{00} = a_0^2 + a_2^2 + a_3^2 + a_5^2$$

$$q_{01} = 2(a_0a_1 + a_3a_4)$$

$$q_{02} = a_1^2 - a_2^2 + a_4^2 - a_5^2$$

$$q_{10} = 2(a_0a_2 + a_3a_5)$$

$$q_{11} = 2(a_1a_2 + a_4a_5)$$

Finally,

$$\lambda^2 = r_0(c_1) + s_1r_1(c_1)$$

where $r_0(z) = \sum_{i=0}^2 r_{0i}z^i$ and $r_1(z) = \sum_{i=0}^1 r_{1i}z$. The coefficients are

$$r_{00} = b_0^2$$

$$r_{01} = 0$$

$$r_{02} = b_3^2 - b_0^2$$

$$r_{10} = 0$$

$$r_{11} = 2b_0b_3$$

Combining these yields

$$\begin{aligned} 0 &= \left[(p_0^2 - r_0q_0) + (1 - c_1^2)(p_1^2 - r_1q_1) \right] + s_1 \left[2p_0p_1 - r_0q_1 - r_1q_0 \right] \\ &= g_0(c_1) + s_1g_1(c_1) \end{aligned} \tag{14.5}$$

where $g_0(z) = \sum_{i=0}^4 g_{0i}z^i$ and $g_1(z) = \sum_{i=0}^3 g_{1i}z^i$. The coefficients are

$$\begin{aligned}
 g_{00} &= p_{00}^2 + p_{10}^2 - q_{00}r_{00} \\
 g_{01} &= 2(p_{00}p_{01} + p_{10}p_{11}) - q_{01}r_{00} - q_{10}r_{11} \\
 g_{02} &= p_{01}^2 + 2p_{00}p_{02} + p_{11}^2 - p_{10}^2 - q_{02}r_{00} - q_{00}r_{02} - q_{11}r_{11} \\
 g_{03} &= 2(p_{01}p_{02} - p_{10}p_{11}) - q_{01}r_{02} + q_{10}r_{11} \\
 g_{04} &= p_{02}^2 - p_{11}^2 - q_{02}r_{02} + q_{11}r_{11} \\
 g_{10} &= 2p_{00}p_{10} - q_{10}r_{00} \\
 g_{11} &= 2(p_{01}p_{10} + p_{00}p_{11}) - q_{11}r_{00} - q_{00}r_{11} \\
 g_{12} &= 2(p_{02}p_{10} + p_{01}p_{11}) - q_{10}r_{02} - q_{01}r_{11} \\
 g_{13} &= 2p_{02}p_{11} - q_{11}r_{02} - q_{02}r_{11}
 \end{aligned}$$

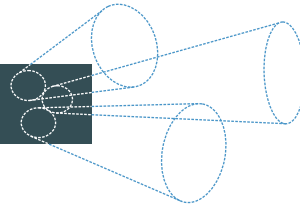
The s_1 term can be eliminated by solving $g_0 = -s_1g_1$ and squaring to obtain

$$0 = g_0^2 - (1 - c_1^2)g_1^2 = h(c_1)$$

where $h(z) = \sum_{i=0}^8 h_i z^i$. The coefficients are

$$\begin{aligned}
 h_0 &= g_{00}^2 - g_{10}^2 \\
 h_1 &= 2(g_{00}g_{01} - g_{10}g_{11}) \\
 h_2 &= g_{01}^2 + g_{10}^2 - g_{11}^2 + 2(g_{00}g_{02} - g_{10}g_{12}) \\
 h_3 &= 2(g_{01}g_{02} + g_{00}g_{03} + g_{10}g_{11} - g_{11}g_{12} - g_{10}g_{13}) \\
 h_4 &= g_{02}^2 + g_{11}^2 - g_{12}^2 + 2(g_{01}g_{03} + g_{00}g_{04} + g_{10}g_{12} - g_{11}g_{13}) \\
 h_5 &= 2(g_{02}g_{03} + g_{01}g_{04} + g_{11}g_{12} + g_{10}g_{13} - g_{12}g_{13}) \\
 h_6 &= g_{03}^2 + g_{12}^2 - g_{13}^2 + 2(g_{02}g_{04} + g_{11}g_{13}) \\
 h_7 &= 2(g_{03}g_{04} + g_{12}g_{13}) \\
 h_8 &= g_{04}^2 + g_{13}^2
 \end{aligned}$$

To find the minimum squared distance, all the real-valued roots of $h(c_1) = 0$ are computed. For each c_1 , compute $s_1 = \pm\sqrt{1 - c_1^2}$ and choose either (or both) of these that satisfies Equation (14.5). For each pair (c_1, s_1) , solve for (c_0, s_0) in Equation (14.4). The main numerical issue to deal with is how close to zero is $\det(M)$.



INTERSECTION METHODS

The distance methods of Chapter 14 may be used to detect intersections of objects. Of course, a distance of zero means the objects are intersecting. The distance functions compute a pair of closest points, but if two objects are overlapping, the number of intersection points may be infinite. This chapter presents some commonly used intersection methods that are not based on distance. The first part is dedicated to line-object intersections and the last part is dedicated to object-object intersections. In all cases, objects are treated as *solids*. A planar component such as a triangle or rectangle and a spatial component such as a sphere or box are defined to be the set of boundary and interior points.

An intersection query is classified as a *test-intersection query* or a *find-intersection query*. The test-intersection query cares only if the two objects intersect, not where. The find-intersection query does involve computing the set of intersection (or a subset of intersection, if this suffices for an application). In many cases, a test-intersection query is less expensive to compute than a find-intersection query.

15.1 LINEAR COMPONENTS AND CONVEX OBJECTS

The test-intersection and find-intersection queries for linear components and objects are all structured to return a Boolean value that is true if and only if there is an intersection. The find-intersection query returns additional information about the intersection set. Generally, this set can be arbitrarily complicated, but we restrict

our attention to convex objects that are bounded; see the introduction to Chapter 8 and Figure 8.1. The convexity and boundedness of the object guarantee that the intersection set of a line and the object is either the empty set, a single point, or a line segment. We may use this fact to compute ray-object and segment-object intersections by first computing the t -values for the line-object intersections, and then refining them based on the t -interval constraints for the ray or segment. The find-intersection queries involving linear components, say, $\mathbf{P} + t\mathbf{D}$, and convex objects may therefore be structured to return an array of q values for t , where $0 \leq q \leq 2$. If $q = 0$, the intersection set is empty and the t array has no valid entries. If $q = 1$, the set of intersection is a single point and the t array has the valid entry $t[0]$. If $q = 2$, the set of intersection is a line segment and the t array has two valid entries $t[0]$ and $t[1]$. It is guaranteed that $t[0] < t[1]$.

Abstractly, let the find-intersection queries have the prototypes

```
bool Find (Line line, Object object, int& quantity, float tvalue[2]);
bool Find (Ray ray, Object object, int& quantity, float tvalue[2]);
bool Find (Segment segment, Object object, int& quantity, float tvalue[2]);
```

All the mathematical analysis goes into the implementation for the line-object query. The ray-object query is implemented as

```
bool Find (Ray ray, Object object, int& quantity, float tvalue[2])
{
    Line line = <convert ray to line>;
    Find(line,object,quantity,tvalue);

    if (quantity == 2)
    {
        // Intersect the line-object interval [t0,t1] with [0,infinity).
        if (tvalue[1] < 0)
        {
            quantity = 0;
        }
        else if (tvalue[1] > 0)
        {
            quantity = 2;
            if (tvalue[0] < 0)
            {
                tvalue[0] = 0;
            }
        }
    }
    else
    {
        quantity = 1;
    }
}
```

```

        tvalue[0] = 0;
    }
}
else if (quantity == 1)
{
    // Intersect the line-object parameter t0 with [0,infinity).
    if (tvalue[0] < 0)
    {
        quantity = 0;
    }
}

return quantity > 0;
}

```

The segment-object query is implemented as

```

bool Find (Segment segment, Object object, int& quantity, float tvalue[2])
{
    Line line = <convert segment to line>;
    Find(line,object,quantity,tvalue));

    if (quantity == 2)
    {
        // Intersect the line-object interval [t0,t1] with [-e,e].
        if (tvalue[0] > segment.e || tvalue[1] < -segment.e)
        {
            quantity = 0;
        }
        else if (tvalue[1] > -segment.e)
        {
            if (tvalue[0] < segment.e)
            {
                if (tvalue[0] < -segment.e)
                {
                    tvalue[0] = -segment.e;
                }

                if (tvalue[1] > segment.e)
                {
                    tvalue[1] = segment.e;
                }

                if (tvalue[0] < tvalue[1])

```

```

        {
            quantity = 2;
        }
        else
        {
            quantity = 1;
        }
    }
    else // tvalue[0] == segment.e
    {
        quantity = 1;
    }
}
else // tvalue[1] == -segment.e
{
    quantity = 1;
    tvalue[0] = tvalue[1];
}
}
else if (quantity == 1)
{
    // Intersect the line-object parameter t0 with [-e,e].
    if (fabs(tvalue[0]) > segment.e)
    {
        quantity = 0;
    }
}

return quantity > 0;
}

```

The next sections describe the various line-object intersection algorithms for the purposes of the find-intersection queries. However, the test-intersection queries in many cases can take advantage of the special nature of the linear component and object, so these are also discussed.

15.2 LINEAR COMPONENT AND PLANAR COMPONENT

The general strategy for computing intersections between a linear component and a planar object is

- Compute the intersection point (if any) of the linear component and plane.
- Test if the intersection point is inside the object. This is a 2D problem.

For triangles or rectangles, the distance methods discussed in Sections 14.4 and 14.6 had subproblems that amounted to computing the intersection points with the plane. These are essentially the idea of [Möl97]. The pseudocode needs only slight modification. Moreover, it is usually the case that triangle or rectangle parameters are not needed at the point of intersection. For example, the line-triangle find-intersection query is

```
bool Find (Line line, Triangle triangle, int& quantity, float tvalue[2])
{
    const float epsilon = <small tolerance, say, 1e-06>;
    Vector3 E0 = triangle.Q[1] - triangle.Q[0];
    Vector3 E1 = triangle.Q[2] - triangle.Q[0];
    Vector3 N = UnitCross(E0,E1); // normalized cross product
    float NdD = Dot(N,line.D);
    if (|NdD| > epsilon)
    {
        // The line and triangle are not parallel. Compute
        // the intersection point.
        Vector3 PmQ0 = line.P - triangle.Q[0];
        float UdE0 = Dot(line.U,E0);
        float UdE1 = Dot(line.U,E1);
        float UdPmQ0 = Dot(line.U,PmQ0);
        float VdE0 = Dot(line.V,E0);
        float VdE1 = Dot(line.V,E1);
        float VdPmQ0 = Dot(line.V,PmQ0);
        float invDet = 1/(UdE0 * VdE1 - UdE1 * VdE0);
        float b1 = (VdE1 * UdPmQ0 - UdE1 * VdPmQ0) * invDet;
        float b2 = (UdE0 * VdPmQ0 - VdE0 * UdPmQ0) * invDet;
        float b0 = 1 - b1 - b2;
        if (b0 >= 0 && b1 >= 0 && b2 >= 0)
        {
            // The intersection point is inside the triangle.
            tvalue[0] = -Dot(N,PmQ0)/NdD;
            return true;
        }
    }

    // What to do here?
    return false;
}
```

If the line and triangle are parallel, the pseudocode reports that there is no intersection. This is correct when the line and triangle are not coplanar. However, if they are coplanar, it is possible that the two objects intersect. In many 3D applications, skipping the coplanar case is usually a safe thing to do. For example, if you were in

the midst of a picking operation, grazing pick-rays are not of much interest. You tend to pick an object by clicking on a pixel well inside the object. Another example is firing a laser gun at an opponent's character. Who is to say that a grazing laser strike really hit the character? Simply require that the character must be struck firmly inside his boundaries. That said, if you want to handle the grazing contact, then

EXERCISE Modify the line-triangle intersection code to handle the case when the line and triangle are coplanar. ■
15.1

Similar pseudocode may be extracted from the distance functions in Sections 14.4 and 14.6 for ray-triangle, segment-triangle, line-rectangle, ray-rectangle, and segment-rectangle. You can also just rely on the generic code mentioned previously that allows you to put the core of the algorithms in the line-object code, and then handle ray-object and segment-object by clamping the line parameter appropriately.

15.3 LINEAR COMPONENT AND ORIENTED BOX

Let the oriented box have center $\mathbf{C}$, orthonormal axis directions $\mathbf{U}_i$, and extents e_i for $0 \leq i \leq 2$. The linear component is $\mathbf{P} + t\mathbf{D}$.

15.3.1 TEST-INTERSECTION QUERY

The test-intersection queries use the method of separating axes; see Section 8.1. This method is based on Minkowski differences of sets; in our current case these are linear components and an oriented box.

Lines and OBBs

The application of the method of separating axes to a line and an OBB is described here. The Minkowski difference of the OBB and the line is an infinite convex polyhedron obtained by extruding the OBB along the line and placing it appropriately in space. Figure 15.1 illustrates this process in two dimensions. Four of the OBB edges are perpendicular to the plane of the page. Two of those edges are highlighted with points and labeled as E_0 and E_1 . The edge E_0 is extruded along the line direction $\mathbf{D}$. The resulting face, labeled F_0 , is an infinite planar strip with a normal vector $\mathbf{U}_0 \times \mathbf{D}$, where $\mathbf{U}_0$ is the unit-length normal of the face of the OBB that is coplanar with the page. The edge E_1 is extruded along the line direction to produce face F_1 . Because edges E_0 and E_1 are parallel, face F_1 also has a normal vector $\mathbf{U}_0$. The maximum number of faces that the infinite polyhedron can have is six (project the OBB onto a plane with normal $\mathbf{D}$ and obtain a hexagon), one for each of the independent OBB edge directions. These directions are the same as the OBB face normal directions,

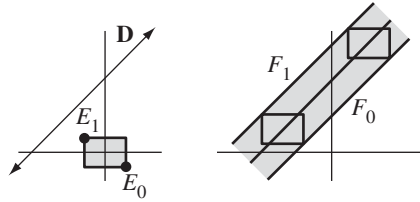


Figure 15.1 An OBB and a line. The OBB is extruded along the line in the direction $\mathbf{D}$. The illustration is based on the OBB having its center at the origin. If the center were not at the origin, the extruded object would be translated from its current location.

so the six faces are partitioned into three pairs of parallel faces with normal vectors $\mathbf{U}_i \times \mathbf{D}$.

Now that we have the potential separating axis directions, the separation tests are

$$|\mathbf{U}_0 \cdot \mathbf{D} \times \Delta| > e_1 |\mathbf{D} \cdot \mathbf{U}_2| + e_2 |\mathbf{D} \cdot \mathbf{U}_1|$$

$$|\mathbf{U}_1 \cdot \mathbf{D} \times \Delta| > e_0 |\mathbf{D} \cdot \mathbf{U}_2| + e_2 |\mathbf{D} \cdot \mathbf{U}_0|$$

$$|\mathbf{U}_2 \cdot \mathbf{D} \times \Delta| > e_0 |\mathbf{D} \cdot \mathbf{U}_1| + e_1 |\mathbf{D} \cdot \mathbf{U}_0|$$

where $\Delta = \mathbf{P} - \mathbf{C}$. The term $\mathbf{U}_i \cdot \mathbf{D} \times \Delta$ is used instead of the mathematically equivalent $\mathbf{U}_i \times \mathbf{D} \cdot \Delta$ in order for the implementation to compute $\mathbf{D} \times \Delta$ once, leading to a reduced operation count for all three separation tests. The implementation of `IntrLine3Box3::Test` using this method is straightforward:

```
bool Test (Line line, Box box)
{
    float AWdU[3], AWxDdU[3], rhs;

    Vector3 diff = line.P - box.C;
    Vector3 WxD = Cross(line.D, diff);

    AWdU[1] = |Dot(line.D, box.U[1])|;
    AWdU[2] = |Dot(line.D, box.U[2])|;
    AWxDdU[0] = |Dot(WxD, box.U[0])|;
    rhs = box.e[1] * AWdU[2] + box.e[2] * AWdU[1];
    if (AWxDdU[0] > rhs)
    {
        return false;
    }
}
```

```

    AWdU[0] = |Dot(line.D, box.U[0])|;
    AWxDdU[1] = |Dot(WxD, box.U[1])|;
    rhs = box.e[0] * AWdU[2] + box.e[2] * AWdU[0];
    if (AWxDdU[1] > rhs)
    {
        return false;
    }

    AWxDdU[2] = |Dot(WxD, box.U[2])|;
    rhs = box.e[0] * AWdU[1] + box.e[1] * AWdU[0];
    if (AWxDdU[2] > rhs)
    {
        return false;
    }

    return true;
}

```

Rays and OBBs

The infinite convex polyhedron that corresponds to the Minkowski difference of a line and an OBB becomes a semi-infinite object in the case of a ray and an OBB. Figure 15.2 illustrates in two dimensions. The semi-infinite convex polyhedron has the same three pairs of parallel faces as for the line, but the polyhedron has the OBB as an end cap. The OBB contributes three additional faces and corresponding normal vectors. Thus, we have six potential separating axes. The separation tests are

$$\begin{aligned}
 |\mathbf{U}_0 \cdot \mathbf{D} \times \Delta| &> e_1 |\mathbf{D} \cdot \mathbf{U}_2| + e_2 |\mathbf{D} \cdot \mathbf{U}_1| \\
 |\mathbf{U}_1 \cdot \mathbf{D} \times \Delta| &> e_0 |\mathbf{D} \cdot \mathbf{U}_2| + e_2 |\mathbf{D} \cdot \mathbf{U}_0| \\
 |\mathbf{U}_2 \cdot \mathbf{D} \times \Delta| &> e_0 |\mathbf{D} \cdot \mathbf{U}_1| + e_1 |\mathbf{D} \cdot \mathbf{U}_0| \\
 |\mathbf{U}_0 \cdot \Delta| &> e_0, & (\mathbf{U}_0 \cdot \Delta)(\mathbf{U}_0 \cdot \mathbf{D}) \geq 0 \\
 |\mathbf{U}_1 \cdot \Delta| &> e_1, & (\mathbf{U}_1 \cdot \Delta)(\mathbf{U}_1 \cdot \mathbf{D}) \geq 0 \\
 |\mathbf{U}_2 \cdot \Delta| &> e_2, & (\mathbf{U}_2 \cdot \Delta)(\mathbf{U}_2 \cdot \mathbf{D}) \geq 0
 \end{aligned}$$

The first three are the same as for a line. The last three use the OBB face normals for the separation tests. To illustrate these tests, see Figure 15.3.

The projected OBB is the finite interval $[-e_i, e_i]$. The projected ray is a semi-infinite interval on the t -axis. The origin is $\Delta \cdot \mathbf{U}_i$, and the direction (a signed scalar) is $\mathbf{D} \cdot \mathbf{U}_i$. The top portion of the figure shows a positive signed direction and an origin that satisfies $\Delta \cdot \mathbf{U}_i > e_i$. The finite interval and the semi-infinite interval are disjoint, in which case the original OBB and ray are separated. If instead the projected ray

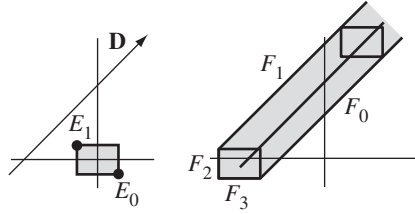


Figure 15.2 An OBB and a ray. The OBB is extruded along the ray in the direction $\mathbf{D}$. The faces F_0 and F_1 are generated by OBB edges and $\mathbf{D}$. The faces F_2 and F_3 are contributed from the OBB.

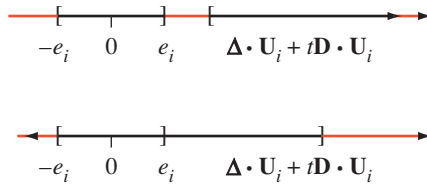


Figure 15.3 Projections of an OBB and a ray onto the line with direction $\mathbf{U}_i$ (a normal to an OBB face). The OBB center $\mathbf{C}$ is subtracted from the OBB as well as the ray origin $\mathbf{P}$. The translated OBB projects to the interval $[-e_i, e_i]$, where e_i is the OBB extent associated with $\mathbf{U}_i$. The translated ray is $\Delta + t\mathbf{D}$, where $\Delta = \mathbf{P} - \mathbf{C}$, and projects to $\mathbf{U}_i \cdot \Delta + t\mathbf{U}_i \cdot \mathbf{D}$.

direction is negative, $\mathbf{D} \cdot \mathbf{U}_i < 0$, the semi-infinite interval overlaps the finite interval. The original OBB and ray are not separated by the axis with direction $\mathbf{U}_i$. Two similar configurations exist when $\Delta \cdot \mathbf{U}_i < -e_i$. The condition $|\mathbf{U}_i \cdot \Delta| > e_i$ says that the projected ray origin is farther away from zero than the projected box extents. The condition $(\mathbf{U}_i \cdot \Delta)(\mathbf{U}_i \cdot \mathbf{D}) > 0$ guarantees that the projected ray points away from the projected OBB.

The implementation of the test-intersection query is

```
bool Test (Ray ray, Box box)
{
    float WdU[3], AWdU[3], DdU[3], ADdU[3], AWxDdU[3], rhs;

    Vector3 diff = ray.P - box.C;
```



```

WdU[0] = Dot(ray.D,box.U[0]);
AWdU[0] = |WdU[0]|;
DdU[0] = Dot(diff,box.U[0]);
ADdU[0] = |DdU[0]|;
if (ADdU[0] > box.e[0] and DdU[0] * WdU[0] >= 0)
{
    return false;
}

WdU[1] = Dot(ray.D,box.U[1]);
AWdU[1] = |WdU[1]|;
DdU[1] = Dot(diff,box.U[1]);
ADdU[1] = |DdU[1]|;
if (ADdU[1] > box.e[1] and DdU[1] * WdU[1] >= 0)
{
    return false;
}

WdU[2] = Dot(ray.D,box.U[2]);
AWdU[2] = |WdU[2]|;
DdU[2] = Dot(diff,box.U[2]);
ADdU[2] = |DdU[2]|;
if (ADdU[2] > box.e[2] and DdU[2] * WdU[2] >= 0)
{
    return false;
}

Vector3 WxD = Cross(ray.D,diff);

AWxDdU[0] = |Dot(WxD,box.U[0])|;
rhs = box.e[1] * AWdU[2] + box.e[2] * AWdU[1];
if (AWxDdU[0] > rhs)
{
    return false;
}

AWxDdU[1] = |Dot(WxD,box.U[1])|;
rhs = box.e[0] * AWdU[2] + box.e[2] * AWdU[0];
if (AWxDdU[1] > rhs)
{
    return false;
}

AWxDdU[2] = |Dot(WxD,box.U[2])|;

```

```

    rhs = box.e[0] * AWdU[1] + box.e[1] * AWdU[0];
    if (AWxDdU[2] > rhs)
    {
        return false;
    }

    return true;
}

```

Segments and OBBs

The semi-infinite convex polyhedron that corresponds to the Minkowski difference of a ray and an OBB becomes a finite object in the case of a segment and an OBB. Figure 15.4 illustrates in two dimensions. The infinite convex polyhedron has the same three pairs of parallel faces as for the ray and line, but the polyhedron has the OBB as an end cap on both ends of the segment. The OBB contributes three additional faces and corresponding normal vectors, a total of six potential separating axes. The separation tests are

$$|\mathbf{U}_0 \cdot \mathbf{D} \times \Delta| > e_1 |\mathbf{D} \cdot \mathbf{U}_2| + e_2 |\mathbf{D} \cdot \mathbf{U}_1|$$

$$|\mathbf{U}_1 \cdot \mathbf{D} \times \Delta| > e_0 |\mathbf{D} \cdot \mathbf{U}_2| + e_2 |\mathbf{D} \cdot \mathbf{U}_0|$$

$$|\mathbf{U}_2 \cdot \mathbf{D} \times \Delta| > e_0 |\mathbf{D} \cdot \mathbf{U}_1| + e_1 |\mathbf{D} \cdot \mathbf{U}_0|$$

$$|\mathbf{U}_0 \cdot \Delta| > e_0 + e |\mathbf{U}_0 \cdot \mathbf{D}|$$

$$|\mathbf{U}_1 \cdot \Delta| > e_1 + e |\mathbf{U}_1 \cdot \mathbf{D}|$$

$$|\mathbf{U}_2 \cdot \Delta| > e_2 + e |\mathbf{U}_2 \cdot \mathbf{D}|$$

where e is the extent of the segment. Figure 15.5 shows a typical separation of the projections on some separating axis. The intervals are separated when $\mathbf{W} \cdot \Delta - e |\mathbf{W} \cdot \mathbf{D}| > r$ or when $\mathbf{W} \cdot \Delta + e |\mathbf{W} \cdot \mathbf{D}| < -r$. These may be combined into a joint statement: $|\mathbf{W} \cdot \Delta| > r + e |\mathbf{W} \cdot \mathbf{D}|$. The implementation of the test-intersection query is

```

bool Test (Segment segment, Box box)
{
    float AWdU[3], ADdU[3], AWxDdU[3], rhs;

    Vector3 diff = segment.P - box.C;

    AWdU[0] = |Dot(segment.D, box.U[0])|;
    ADdU[0] = |Dot(diff, box.U[0])|;
    rhs = box.e[0] + segment.e * AWdU[0];

```

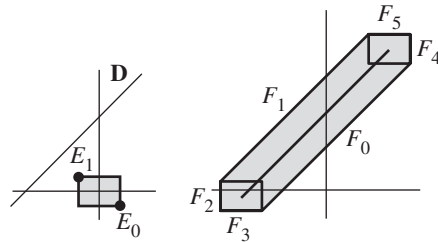


Figure 15.4 An OBB and a segment. The OBB is extruded along the segment in the direction $\mathbf{D}$. The faces F_0 and F_1 are generated by OBB edges and $\mathbf{D}$. The faces F_2 and F_3 are contributed from the OBB at one endpoint; the faces F_4 and F_5 are contributed from the OBB at the other endpoint.

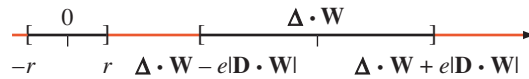


Figure 15.5 Projections of an OBB and a segment onto a line with direction $\mathbf{W}$. The OBB center $\mathbf{C}$ is subtracted from the OBB as well as the segment origin $\mathbf{P}$. The translated OBB projects to an interval $[-r, r]$. The translated segment is $\Delta + t\mathbf{D}$, where $\Delta = \mathbf{P} - \mathbf{C}$, and projects to $\mathbf{W} \cdot \Delta + t\mathbf{W} \cdot \mathbf{D}$.

```

if (ADdU[0] > rhs)
{
    return false;
}

AWdU[1] = |Dot(segment.D, box.U[1])|;
ADdU[1] = |Dot(diff, box.U[1])|;
rhs = box.e[1] + segment.e * AWdU[1];
if (ADdU[1] > rhs)
{
    return false;
}

AWdU[2] = |Dot(segment.D, box.U[2])|;
ADdU[2] = |Dot(diff, box.U[2])|;

```

```

rhs = box.e[2] + segment.e * AWdU[2];
if (ADdU[2] > rhs)
{
    return false;
}

Vector3 WxD = Cross(segment.D,diff);

AWxDdU[0] = |Dot(WxD,box.U[0])|;
rhs = box.e[1] * AWdU[2] + box.e[2] * AWdU[1];
if (AWxDdU[0] > rhs)
{
    return false;
}

AWxDdU[1] = |Dot(WxD,box.U[1])|;
rhs = box.e[0] * AWdU[2] + box.e[2] * AWdU[0];
if (AWxDdU[1] > rhs)
{
    return false;
}

AWxDdU[2] = |Dot(WxD,box.U[2])|;
rhs = box.e[0] * AWdU[1] + box.e[1] * AWdU[0];
if (AWxDdU[2] > rhs)
{
    return false;
}

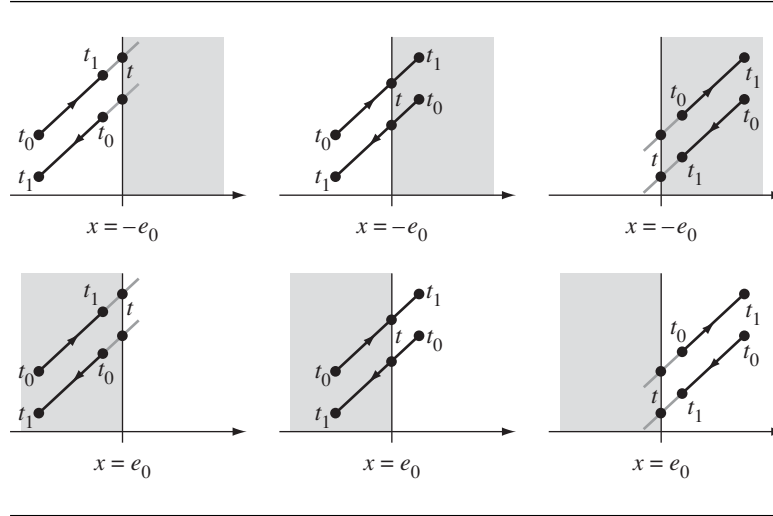
return true;
}

```

15.3.2 FIND-INTERSECTION QUERY

The line-box find-intersection query is based on Liang-Barsky clipping [LB84, FvDFH90] of parametric lines against the box faces one at a time. The idea is to start with a t interval $[t_0, t_1]$ representing the current linear component. Initially, the interval is infinite: $t_0 = -\infty$ and $t_1 = \infty$. The line is converted to the box coordinate system. The line origin $\mathbf{P}$ is mapped to $\mathbf{P}' = (x_p, y_p, z_p)$ in the box coordinate system via

$$\mathbf{P} = \mathbf{C} + x_p \mathbf{U}_0 + y_p \mathbf{U}_1 + z_p \mathbf{U}_2$$

Figure 15.6 Line clipping against the x -faces of the box.

Thus, $x_p = \mathbf{U}_0 \cdot (\mathbf{P} - \mathbf{C})$, $y_p = \mathbf{U}_1 \cdot (\mathbf{P} - \mathbf{C})$, and $z_p = \mathbf{U}_2 \cdot (\mathbf{P} - \mathbf{C})$. The line direction $\mathbf{D}$ is mapped to $\mathbf{D}' = (x_d, y_d, z_d)$ in the box coordinate system via

$$\mathbf{D} = x_d \mathbf{U}_0 + y_d \mathbf{U}_1 + z_d \mathbf{U}_2$$

Thus, $x_d = \mathbf{U}_0 \cdot \mathbf{D}$, $y_p = \mathbf{U}_1 \cdot \mathbf{D}$, and $z_d = \mathbf{U}_2 \cdot \mathbf{D}$. In the box coordinate system, the box is naturally axis aligned. If (x, y, z) is a box point, then $|x| \leq e_0$, $|y| \leq e_1$, and $|z| \leq e_2$.

Figure 15.6 illustrates the clipping of the line $(x_p, y_p, z_p) + t(x_d, y_d, z_d)$ against the x -faces of the box. The top three images in the figure show clipping against the face $x = -e_0$, and the bottom three images show clipping against the face $x = e_0$. The clipping algorithm depends on the orientation of the line's direction vector relative to the x -axis. For a plane $x = a$, the intersection point of the line and plane is determined by

$$x_p + tx_d = a$$

We need to solve for t . The three cases are $x_d > 0$, $x_d < 0$, and $x_d = 0$.

Consider $x_d > 0$. In Figure 15.6, the linear components of interest are those with the arrow showing the components pointing generally in the positive x -direction. The t -value for the intersection is $t = (-e_0 - x_p)/x_d$. The decision on clipping is illustrated in the pseudocode:

```

if (t > t1) then
    cull the linear component; // Figure 15.6 (a)
else if (t > t0) then
    t0 = t; // Figure 15.6 (b)
else
    do nothing; // Figure 15.6 (c)

```

If $x_d < 0$, the relevant linear components in Figure 15.6 are those with the arrow showing the components pointing generally in the negative x -direction. The pseudocode is

```

if (t < t0) then
    cull the linear component; // Figure 15.6 (a)
else if (t < t1) then
    t1 = t; // Figure 15.6 (b)
else
    do nothing; // Figure 15.6 (c)

```

Finally, if $x_d = 0$, the linear component is parallel to the x -faces. The component is either outside the box, in which case it is culled, or inside or on the box, in which case no clipping against the x -faces needs to be performed. The pseudocode is

```

n = -e0 - xp;
if (n > 0) then
    cull the linear component;
else
    do nothing;

```

A similar construction applies for the face $x = e_0$, except that the sense of direction is reversed. The bottom three images of Figure 15.6 apply here. The equation to solve is $x_p + tx_d = e_0$. If $x_d \neq 0$, the solution is $t = (e_0 - x_p)/t_d$. If $x_d > 0$, the pseudocode is

```

if (t < t0) then
    cull the linear component; // Figure 15.6 (f)
else if (t < t1) then
    t1 = t; // Figure 15.6 (e)
else
    do nothing; // Figure 15.6 (d)

```

If $x_d < 0$, the pseudocode is

```

if (t > t1) then
    cull the linear component; // Figure 15.6 (f)
else if (t > t0) then
    t0 = t; // Figure 15.6 (e)
else
    do nothing; // Figure 15.6 (d)

```

If $x_d = 0$, the linear component is parallel to the x -faces. The component is either outside the box, in which case it is culled, or inside or on the box, in which case no clipping against the x -faces needs to be performed. The pseudocode is

```

n = e0 - xp;
if (n < 0) then
    cull the linear component;
else
    do nothing;

```

A goal of the construction is to keep the clipping code to a minimum. With this in mind, notice that the clipping code for the case $x = e_0$ and $x_d > 0$ is the same as that for the case $x = -e_0$ and $x_d < 0$. The clipping code for the case $x = e_0$ and $x_d < 0$ is the same as that for the case $x = -e_0$ and $x_d > 0$. The cases when $x_d = 0$ differ only by the test of n —in one case tested for positivity, in the other for negativity. We may consolidate the six code blocks into three by processing the $x = e_0$ cases using $-x_d$ and $-n = x_p - e_0$. The following pseudocode shows the consolidated code. One optimization occurs: a division is performed to compute t only if the linear component is not culled.

```

bool Clip(float denom, float numer, float& t0, float& t1)
{
    if (denom > 0)
    {
        if (numer > denom * t1)
            return false;
        if (numer > denom * t0)
            t0 = numer / denom;
        return true;
    }

    if (denom < 0)
    {
        if (numer > denom * t0)
            return false;
    }
}

```

```

        if (number > denom * t1)
            t1 = number / denom;
        return true;
    }

    return number <= 0;
}

```

The return value is false if the current linear component is culled; otherwise, the return value is true, indicating that the linear component was clipped or just kept as is.

The clipping against all the faces is encapsulated by the line-box find-intersection query function.

```

bool Find (Line line, Box box, int& quantity, float tvalue[2])
{
    // Convert line to box coordinates.
    Point delta = line.P - box.C;
    Point P, D;
    for (i = 0; i < 3; i++)
    {
        P[i] = Dot(delta, box.U[i]);
        D[i] = Dot(line.D, box.U[i]);
    }

    float t0 = -infinity, t1 = +infinity;
    bool notCulled =
        Clip(+D.x, -P.x - box.e[0], t0, t1) &&
        Clip(-D.x, +P.x - box.e[0], t0, t1) &&
        Clip(+D.y, -P.y - box.e[1], t0, t1) &&
        Clip(-D.y, +P.y - box.e[1], t0, t1) &&
        Clip(+D.z, -P.z - box.e[2], t0, t1) &&
        Clip(-D.z, +P.z - box.e[2], t0, t1);

    if (notCulled)
    {
        if (t1 > t0)
        {
            quantity = 2;
            tvalue[0] = t0;
            tvalue[1] = t1;
        }
    }
}

```



```

        else
        {
            quantity = 1;
            tvalue[0] = t0;
        }
    }
    else
    {
        quantity = 0;
    }

    return quantity > 0;
}

```

15.4 LINEAR COMPONENT AND SPHERE

Both the test-intersection and find-intersection queries are discussed here.

15.4.1 LINE AND SPHERE

Consider the general case of intersection between a line and a sphere. A sphere with center $\mathbf{C}$ and radius r is specified by $|\mathbf{X} - \mathbf{C}|^2 - r^2 = 0$. Replacing $\mathbf{X}$ by $\mathbf{P} + t\mathbf{D}$ leads to the quadratic equation

$$0 = |t\mathbf{D} + \mathbf{P} - \mathbf{C}|^2 - r^2 = t^2 + 2t\mathbf{D} \cdot (\mathbf{P} - \mathbf{C}) + |\mathbf{P} - \mathbf{C}|^2 - r^2 = t^2 + 2a_1t + a_0 = q(t)$$

The quadratic formula may be used to solve the equation formally:

$$t = -a_1 \pm \sqrt{a_1^2 - a_0}$$

Let $\Delta = a_1^2 - a_0$, called the *discriminant* of the quadratic equation. The classification of intersection is

- $\Delta < 0$. The roots are complex-valued, so the line does not intersect the sphere.
- $\Delta = 0$. The equation has a repeated real-valued root, so the line intersects the sphere in a single point. Necessarily the line is tangent to the sphere.
- $\Delta > 0$. The equation has two distinct real-valued roots, so the line intersects the sphere in two points. The intersections are transverse in the sense that the line

passes from outside to inside the sphere and inside to outside the sphere at the two points.

Regarding the test-intersection query, we could just directly apply the quadratic formula and classify the roots to determine the intersection of linear components and spheres, but some additional analysis allows us to avoid the expensive square root calculation in that formula.

The sign of $a_0 = |\mathbf{P} - \mathbf{C}|^2 - r^2$ determines whether or not $\mathbf{P}$ is inside the sphere ($a_0 < 0$), on the sphere ($a_0 = 0$), or outside the sphere ($a_0 > 0$). This allows a quick out in the test-intersection query: If $a_0 \leq 0$, then the line intersects the sphere because $\mathbf{P}$ is inside or on the sphere.

```
bool Test (Line line, Sphere sphere)
{
    Point delta = line.P - sphere.C;
    float a0 = Dot(delta, delta) - sphere.r * sphere.r;
    if (a0 <= 0)
    {
        // line.P is inside or on the sphere.
        return true;
    }
    // Else: line.P is outside the sphere.

    float a1 = Dot(line.D, delta);
    float discr = a1 * a1 - a0;
    return discr >= 0;
}
```

The find-intersection query is a straightforward implementation of the construction of the quadratic roots.

```
bool Find (Line line, Sphere sphere, int& quantity, float tvalue[2])
{
    Point delta = line.P - sphere.C;
    float a0 = Dot(delta, delta) - sphere.r * sphere.r;
    float a1 = Dot(line.D, delta);
    float discr = a1 * a1 - a0;

    if (discr < 0)
    {
        // two complex-valued roots, no intersections
        quantity = 0;
    }
}
```

```

        else if (discr >= epsilon)
        {
            // two distinct real-valued roots, two intersections
            float root = sqrt(discr);
            tvalue[0] = -a1 - root;
            tvalue[1] = -a1 + root;
            quantity = 2;
        }
        else // discr is effectively zero.
        {
            // one repeated real-valued root, one intersection
            tvalue[0] = -a1;
            quantity = 1;
        }
        return quantity > 0;
    }
}

```

15.4.2 RAY AND SPHERE

The test-intersection query for a ray versus a sphere is similar to the test-intersection query for a line versus a sphere, except that there is an additional quick out. If the ray origin is inside the sphere ($a_0 \leq 0$), then the ray intersects the sphere. Otherwise, the ray origin is outside the sphere. It is possible that as you move along the ray, you are moving away from the sphere, in which case there is no intersection when $t \geq 0$. This geometric condition is equivalent to the squared distance $q(t)$ increasing for all $t \geq 0$. To be increasing, the derivative $q'(t) = 2(t + a_1)$ must satisfy $q'(t) \geq 0$. It is sufficient for $a_1 \geq 0$ to guarantee that this happens. The condition $a_1 \geq 0$ is therefore a quick out for a no-intersection result.

```

bool Test (Ray ray, Sphere sphere)
{
    Point delta = ray.P - sphere.C;
    float a0 = Dot(delta, delta) - sphere.r * sphere.r;
    if (a0 <= 0)
    {
        // ray.P is inside or on the sphere.
        return true;
    }
    // Else: ray.P is outside the sphere.

    float a1 = Dot(ray.D, delta);
    if (a1 >= 0)

```

```

{
    // Ray is directed away from the sphere.
    return false;
}

float discr = a1 * a1 - a0;
return discr >= 0;
}

```

15.4.3 SEGMENT AND SPHERE

The test-intersection query for a segment versus sphere is slightly more complicated algebraically than that of ray or line versus sphere, but the end result requires only a minimum of calculation time. If $a_0 \leq 0$, then the segment center $\mathbf{P}$ is inside the sphere and we have an intersection. Otherwise, $\mathbf{P}$ is outside the sphere. If $\Delta = a_1^2 - a_0 < 0$, the line containing the segment does not intersect the sphere, so neither does the segment. If both of these tests do not provide a quick out, we have $q(0) = a_0 > 0$, and $q(t)$ has at least one real-valued root. If $q(e) \leq 0$ or $q(-e) \leq 0$, then $q(t)$ has a root on the interval $[-e, e]$, and the segment intersects the sphere. Otherwise, one of the four configurations must occur as shown in Figure 15.7.

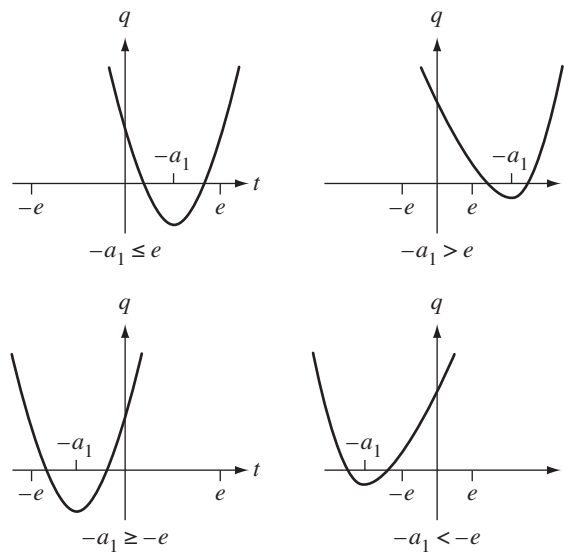


Figure 15.7 The four configurations for the graph of $q(t)$.

Notice that $a_1 = q'(0)$, which is the slope of the graph of $q(t)$ at $t = 0$. The minimum point of the graph occurs when $q'(t) = 2(t + a_1) = 0$, namely, at the time $T = -a_1$. The top two configurations occur when $T = -a_1 > 0$, the bottom two when $T = -a_1 < 0$. The test for intersection may be structured as

```

if (a1 < 0)
{
    // Evaluate q(e).
    qp = segment.e * segment.e + 2 * a1 * segment.e + a0;

    // q(e) <= 0 or T <= e
    if (qp <= 0 || -a1 <= segment.e)
        return true;
}
else
{
    // evaluate q(-e)
    qm = segment.e * segment.e - 2 * a1 * segment.e + a0;

    // q(-e) <= 0 or T >= -e
    if (qm <= 0 || -a1 >= -segment.e)
        return true;
}
return false;

```

However, these may be combined into a smaller number of lines of code. The full test-intersection query is

```

bool Test (Segment segment, Sphere sphere)
{
    Point delta = segment.P - sphere.C;
    float a0 = Dot(delta, delta) - sphere.r * sphere.r;
    if (a0 <= 0)
    {
        // P is inside or on the sphere.
        return true;
    }
    // Else: P is outside the sphere.

    float a1 = Dot(segment.D, delta);
    float discr = a1 * a1 - a0;
    if (discr < 0)
    {
        // two complex-valued roots, no intersections
    }
}

```

```

        return false;
    }

    float absA1 = fabs(a1);
    float qval = segment.e * (segment.e - 2 * absA1) + a0;
    return qval <= 0 || absA1 <= segment.e;
}

```

15.5 LINE AND SPHERE-SWEPT VOLUME

The only sphere-swept volumes of interest here are capsules and lozenges. The test-intersection queries are trivial, but the find-intersection queries involve tedious details when implementing them.

15.5.1 LINE AND CAPSULE

The line has the representation $\mathbf{P} + t\mathbf{D}$. The capsule segment has center $\mathbf{C}$, unit-length direction $\mathbf{W}$, and extent e . Let the capsule radius be r . A coordinate system may be associated with the capsule. The origin of the system is $\mathbf{C}$ and one axis has direction $\mathbf{W}$. Choose two more axis direction vectors $\mathbf{U}$ and $\mathbf{V}$ so that the set $\{\mathbf{U}, \mathbf{V}, \mathbf{D}\}$ is a right-handed orthonormal set. That is, the vectors are unit length and mutually orthogonal, and $\mathbf{W} = \mathbf{U} \times \mathbf{V}$. Any point $\mathbf{P}$ is written in this coordinate system as

$$\mathbf{P} = \mathbf{C} + x\mathbf{U} + y\mathbf{V} + z\mathbf{W}$$

Within this coordinate system, the capsule has a cylinder wall defined by $x^2 + y^2 = r^2$ with $|z| \leq e$. It also has two hemispherical caps, the top cap defined by $x^2 + y^2 + (z - e)^2 = r^2$ for $z \geq e$ and the bottom cap defined by $x^2 + y^2 + (z + e)^2 = r^2$ for $z \leq -e$.

The line-capsule test-intersection query is conceptually easy. The line intersects the capsule whenever the distance between the line and the capsule segment is smaller or equal to the capsule radius. For computational efficiency in avoiding square roots, the squared distance and the squared radius are compared.

```

bool Test (Line line, Capsule capsule)
{
    float sqrDistance = SqrDistance(line, capsule.segment);
    float sqrRadius = capsule.r * capsule.r;
    return sqrDistance <= sqrRadius;
}

```

See Section 14.2.3 for the details about the line-segment squared-distance calculator.

The ray-capsule test-intersection query is similar to the line-capsule query. The ray and capsule intersect if and only if the distance between the ray and the capsule segment is less than or equal to the capsule radius. For computational efficiency in avoiding square roots, the squared distance and the squared radius are compared.

```
bool Test (Ray ray, Capsule capsule)
{
    float sqrDistance = SqrDistance(ray, capsule.segment);
    float sqrRadius = capsule.r * capsule.r;
    return sqrDistance <= sqrRadius;
}
```

The segment-capsule test-intersection query is also similar to the line-capsule query. The segment and capsule intersect if and only if the distance between the segment and the capsule segment is less than or equal to the capsule radius. For computational efficiency in avoiding square roots, the squared distance and the squared radius are compared.

```
bool Test (Segment segment, Capsule capsule)
{
    float sqrDistance = SqrDistance(segment, capsule.segment);
    float sqrRadius = capsule.r * capsule.r;
    return sqrDistance <= sqrRadius;
}
```

The line-capsule find-intersection query is somewhat lengthy to describe. At a high level, the line origin and direction are converted to the coordinate system of the capsule. If the line is parallel (or nearly parallel) to the capsule axis, the intersections of the line with the capsule, if any, must occur on the hemispherical caps. If the line is not parallel to the capsule axis, intersections are computed with the infinite cylinder. If such a point of intersection is within the range $|z| \leq e$, the point is on the capsule's cylinder wall. If we find two such points, the intersection query is finished. If not, an attempt is made to compute the intersection of the line with each of the hemispherical caps. At any time in the construction, if we have two intersection points, the query terminates.

The line origin $\mathbf{P}$ is mapped to $\mathbf{P}' = (x_p, y_p, z_p)$ in the capsule coordinate system via

$$\mathbf{P} = \mathbf{C} + x_p \mathbf{U} + y_p \mathbf{V} + z_p \mathbf{W}$$

where $x_p = \mathbf{U} \cdot (\mathbf{P} - \mathbf{C})$, $y_p = \mathbf{V} \cdot (\mathbf{P} - \mathbf{C})$, and $z_p = \mathbf{W} \cdot (\mathbf{P} - \mathbf{C})$. The line direction $\mathbf{D}$ is mapped to $\mathbf{D}' = (x_d, y_d, z_d)$ in the capsule coordinate system via

$$\mathbf{D} = x_d \mathbf{U} + y_d \mathbf{V} + z_d \mathbf{W}$$

where $x_d = \mathbf{U} \cdot \mathbf{D}$, $y_d = \mathbf{V} \cdot \mathbf{D}$, and $z_d = \mathbf{W} \cdot \mathbf{D}$.

Line Parallel to Capsule Axis

The line is parallel to the capsule axis when $|z_d| = 1$. For numerical robustness, choose a small threshold $\varepsilon > 0$ and decide that the parallel case occurs when $|z_d| \geq 1 - \varepsilon$. The line intersects the capsule when it is contained by the infinite solid cylinder bounding the capsule. This cylinder is $x^2 + y^2 \leq r^2$. An intersection occurs, therefore, when $x_p^2 + y_p^2 \leq r^2$. In the event this condition is satisfied, the two points of intersection are on the hemispherical caps. The intersection with the top hemispherical cap is (x_p, y_p, z_1) , where $x_p^2 + y_p^2 + (z_1 - e)^2 = r^2$ with $z_1 \geq e$. The solution is

$$z_1 = e + \sqrt{r^2 - x_p^2 - y_p^2}$$

The intersection with the bottom hemispherical cap is (x_p, y_p, z_0) , where $x_p^2 + y_p^2 + (z_0 + e)^2 = r^2$ with $z_0 \leq -e$. The solution is

$$z_0 = -e - \sqrt{r^2 - x_p^2 - y_p^2}$$

The find-intersection query needs to compute the t -values corresponding to the line. In the capsule coordinate system, the line is parameterized by $(x_p, y_p, z_p) + t(0, 0, \pm 1)$. If $z_d = 1$, then the intersection with the bottom hemispherical cap satisfies

$$\begin{aligned} (x_p, y_p, z_p) + t_0(0, 0, 1) &= (x_p, y_p, z_0) \rightarrow t_0 = z_0 - z_p \\ &= -z_p - \left(e + \sqrt{r^2 - x_p^2 - y_p^2} \right) \end{aligned}$$

The intersection with the top hemispherical cap satisfies

$$\begin{aligned} (x_p, y_p, z_p) + t_1(0, 0, 1) &= (x_p, y_p, z_1) \rightarrow t_1 = z_1 - z_p \\ &= -z_p + \left(e + \sqrt{r^2 - x_p^2 - y_p^2} \right) \end{aligned}$$

The t -values are chosen so that $t_0 < t_1$. If $z_d = -1$, the intersection with the top hemispherical cap satisfies

$$\begin{aligned} (x_p, y_p, z_p) + t_0(0, 0, -1) &= (x_p, y_p, z_1) \rightarrow t_0 = z_p - z_1 \\ &= z_p - \left(e + \sqrt{r^2 - x_p^2 - y_p^2} \right) \end{aligned}$$

The intersection with the bottom hemispherical cap satisfies

$$\begin{aligned}(x_p, y_p, z_p) + t_1(0, 0, -1) &= (x_p, y_p, z_0) \rightarrow t_1 = z_p - z_0 \\ &= z_p + \left(e + \sqrt{r^2 - x_p^2 - y_p^2}\right)\end{aligned}$$

The t -values are chosen so that $t_0 < t_1$.

Line Not Parallel to Capsule Axis

The parametric line in capsule coordinates is $(x_p, y_p, z_p) + t(x_d, y_d, z_d)$. The intersection of this line with the infinite cylinder amounts to solving a quadratic equation $x^2 + y^2 = r^2$ with $x = x_p + tx_d$ and $y = y_p + ty_d$. The equation is

$$a_2 t^2 + 2a_1 t + a_0 = (x_d^2 + y_d^2)t^2 + 2(x_p x_d + y_p y_d)t + (x_p^2 + y_p^2 - r^2) = 0$$

We know that $a_2 > 0$ since the line is not parallel to the capsule axis. For if it were the case that $a_2 = 0$, then $x_d = y_d = 0$ and $|z_d| = 1$, a contradiction to the nonparallelism. From a numerical perspective, if a_2 is nearly zero, that can cause robustness problems in the root finding. We had tested parallelism by $|z_d| \geq 1 - \varepsilon$ for a small $\varepsilon > 0$. In the nonparallel case, we know $|z_d| < \varepsilon$, in which case

$$a_2 = x_d^2 + y_d^2 = 1 - z_d^2 > 1 - (1 - \varepsilon)^2 = \varepsilon(2 - \varepsilon) \doteq 2\varepsilon$$

As long as the root finder is not susceptible to problems with a leading coefficient on the order of 2ε , the intersection algorithm should behave correctly.

The implementation uses the quadratic equation for root finding:

$$t = \frac{-a_1 \pm \sqrt{a_1^2 - a_0 a_2}}{a_2}$$

Define $\Delta = a_1^2 - a_0 a_2$. If $\Delta < 0$, the line does not intersect the infinite cylinder, and consequently it cannot intersect the capsule's cylinder wall or hemispherical caps. If $\Delta \geq 0$, the line intersects the infinite cylinder in one or two points. However, these points need to be checked such that their z -values satisfy $|z| \leq e$. If T is a root of the quadratic equation, then the z -value for the intersection point is $z = z_p + Tz_d$. If $|z_p + Tz_d| \leq e$, then T is included in the array of parameter values to be returned to the caller. If $\Delta > 0$ and there are two points of intersection satisfying this condition, there is no need to compare the line against the hemispherical caps.

If we have not yet found two intersection points at this time, we need to test for line-hemisphere intersection. The bottom hemisphere is $x^2 + y^2 + (z + e)^2 - r^2 = 0$ for $z \leq -e$. Substituting the parametric line equation into this equation:

$$\begin{aligned}
0 &= (x_p + tx_d)^2 + (y_p + ty_d)^2 + (z_p + e + tz_d)^2 - r^2 \\
&= t^2 + 2(x_px_d + y_py_d + (z_p + e)z_d)t + (x_p^2 + y_p^2 + (z_p + e)^2 - r^2) \\
&= t^2 + 2a_1t + a_0
\end{aligned}$$

The quadratic equation is applied to compute the roots:

$$t = -a_1 \pm \sqrt{a_1^2 - a_0}.$$

If T is a root, the corresponding hemisphere z -value is $z = z_p + Tz_d$. If $z \leq -e$, then T is stored in the array of t -values to be returned to the caller.

If two intersection points are still not found at this time, we test for intersection with the top hemisphere. The equation for the hemisphere is $x^2 + y^2 + (z - e)^2 - r^2 = 0$ for $z \geq e$. Substituting the parametric line equation into this equation:

$$\begin{aligned}
0 &= (x_p + tx_d)^2 + (y_p + ty_d)^2 + (z_p - e + tz_d)^2 - r^2 \\
&= t^2 + 2(x_px_d + y_py_d + (z_p - e)z_d)t + (x_p^2 + y_p^2 + (z_p - e)^2 - r^2) \\
&= t^2 + 2a_1t + a_0
\end{aligned}$$

The quadratic equation is applied to compute the roots:

$$t = -a_1 \pm \sqrt{a_1^2 - a_0}.$$

If T is a root, the corresponding hemisphere z -value is $z = z_p + Tz_d$. If $z \geq e$, then T is stored in the array of t -values to be returned to the caller.

If there are two t -values ready to be returned to the caller, they should be sorted as $t[0] < t[1]$. The order of checking intersections with the infinite cylinder, the bottom hemisphere, and the top hemisphere does not guarantee that the t -values will be sorted.

An implementation of this algorithm is lengthy, but is provided on the CD-ROM for the book.

EXERCISE 15.2

A small improvement can be made to the algorithm. The selection of which hemisphere to test should be made by looking at the z -values of the intersection of the line with the infinite cylinder. For example, if such a z -value satisfies $z > e$, and if the line does intersect a capsule hemisphere, it must do so on the top hemisphere. If the other z -value also satisfies $z > e$, then there is no need to test the bottom hemisphere. But if the other z -value satisfies $z < -e$, then you must test the bottom hemisphere. Modify the line-capsule intersection code to use this improvement. ■

15.5.2 LINE AND LOZENGE

The lozenge's rectangle has center point $\mathbf{C}$, unit-length directions $\mathbf{D}_0$ and $\mathbf{D}_1$ that are perpendicular, and extents e_0 and e_1 . The lozenge radius is r . A third vector is $\mathbf{N} = \mathbf{D}_0 \times \mathbf{D}_1$ and is used with the center and directions to form a coordinate system for the lozenge. Any point $\mathbf{Q}$ may be written as

$$\mathbf{Q} = \mathbf{C} + x\mathbf{D}_0 + y\mathbf{D}_1 + z\mathbf{N}$$

Within this coordinate system, the lozenge is bounded by two planes, four truncated half-cylinders, and four quarter-spheres. The plane equations are $z = \pm r$. The cylinder equations are

$$(x - e_0)^2 + z^2 = r^2, \quad x \geq e_0, |z| \leq r$$

$$(x + e_0)^2 + z^2 = r^2, \quad x \leq -e_0, |z| \leq r$$

$$(y - e_1)^2 + z^2 = r^2, \quad y \geq e_1, |z| \leq r$$

$$(y + e_1)^2 + z^2 = r^2, \quad y \leq -e_1, |z| \leq r$$

The sphere equations are

$$(x - e_0)^2 + (y - e_1)^2 + z^2 = r^2, \quad x \geq e_0, \quad y \geq e_1$$

$$(x + e_0)^2 + (y - e_1)^2 + z^2 = r^2, \quad x \leq -e_0, \quad y \geq e_1$$

$$(x - e_0)^2 + (y + e_1)^2 + z^2 = r^2, \quad x \geq e_0, \quad y \leq -e_1$$

$$(x + e_0)^2 + (y + e_1)^2 + z^2 = r^2, \quad x \leq -e_0, \quad y \leq -e_1$$

The line is $\mathbf{P} + t\mathbf{D}$. It is converted to lozenge coordinates $\mathbf{P}' = (x_p, y_p, z_p)$ and $\mathbf{D}' = (x_d, y_d, z_d)$, just as we did for capsules. If the line $\mathbf{P}' + t\mathbf{D}'$ intersects the lozenge, it does so either on the two planes, the four truncated half-cylinders, or the four quarter-spheres. An implementation is similar to that for a line and capsule, but it is a tedious exercise.

EXERCISE
15.3

Implement the find-intersection query for a line and a lozenge. ■

The test-intersection queries for linear components versus a lozenge are trivial, just as they were for capsules.

```
bool Test (Line line, Lozenge lozenge)
{
    float sqrDistance = SqrDistance(line, lozenge.rectangle);
    float sqrRadius = lozenge.r * lozenge.r;
    return sqrDistance <= sqrRadius;
}
```

```

bool Test (Ray ray, Lozenge lozenge)
{
    float sqrDistance = SqrDistance(ray,lozenge.rectangle);
    float sqrRadius = lozenge.r * lozenge.r;
    return sqrDistance <= sqrRadius;
}

bool Test (Segment segment, Lozenge lozenge)
{
    float sqrDistance = SqrDistance(segment,lozenge.rectangle);
    float sqrRadius = lozenge.r * lozenge.r;
    return sqrDistance <= sqrRadius;
}

```

15.6 LINE AND QUADRIC SURFACE

Generally, a quadric surface is implicitly defined by a quadratic equation $\mathbf{X}^T \mathbf{A} \mathbf{X} + \mathbf{B}^T \mathbf{X} + c = 0$. If the line is $\mathbf{X}(t) = \mathbf{P} + t\mathbf{D}$, substituting it into the quadratic equation of three variables produces an equation in the parameter t :

$$0 = k_2 t^2 + 2k_1 t + k_0 = (\mathbf{D}^T \mathbf{A} \mathbf{D}) t^2 + \mathbf{D}^T (2\mathbf{A} \mathbf{P} + \mathbf{B}) + (\mathbf{P}^T \mathbf{A} \mathbf{P} + \mathbf{B}^T \mathbf{P} + c)$$

The coefficient k_2 may or may not be zero, depending on the type of surface the original equation defines.

15.6.1 LINE AND ELLIPSOID

When the original quadratic equation represents an ellipsoid, it is guaranteed that $k_2 \neq 0$, so the t -equation is quadratic itself. The find-intersection query is similar to that of a line-sphere query. If the ellipsoid is in standard form,

$$(\mathbf{X} - \mathbf{C})^T \mathbf{M} (\mathbf{X} - \mathbf{C}) = 1$$

the quadratic equation in t is

$$(\mathbf{D}^T \mathbf{M} \mathbf{D}) t^2 + 2 (\mathbf{D}^T \mathbf{M} (\mathbf{P} - \mathbf{C})) t + ((\mathbf{P} - \mathbf{C})^T \mathbf{M} (\mathbf{P} - \mathbf{C}) - 1) = 0$$

It is simple enough to solve for t using the quadratic formula. If there are no real-valued roots, the line does not intersect the ellipsoid. If there is a single real-valued root, the line is tangent to the ellipsoid (a single point of intersection). If there are two distinct real-valued roots, the line intersects the ellipsoid in two points.

15.6.2 LINE AND CYLINDER

The cylinders here are assumed to be finite with center $\mathbf{C}$, axis direction $\mathbf{W}$, height h (extent $h/2$), and radius r . The line-cylinder find-intersection query has a similar structure to the line-capsule find-intersection query. For capsules, the algorithm required computing intersections of the line with hemispheres. For cylinders, that part of the algorithm is replaced by computing intersections of lines with the planar disks that cap the ends of the cylinder. The hemispheres were defined by $x^2 + y^2 + (z \pm e)^2 = r^2$. Substituting in the line equation led to a quadratic equation in the line parameter t . The planar disks are $z = \pm r$ with $x^2 + y^2 \leq r^2$, so you need only compute the intersection of the line with the z -planes and test that those points are inside the disks.

If your cylinder is infinite and you care only about the test-intersection query, let the line be represented in cylinder coordinates by $\mathbf{P}' = (x_p, y_p, z_p)$ and $\mathbf{D}' = (x_d, y_d, z_d)$. The line does not intersect the cylinder $x^2 + y^2 \leq r^2$ when the distance from (x_p, y_p) to $(0, 0)$ is larger than r .

15.6.3 LINE AND CONE

In Section 10.4.4, we saw that the double-sided cone is defined by the quadratic equation

$$(\mathbf{X} - \mathbf{C})^T(\mathbf{A}\mathbf{A}^T - (\cos \theta)^2 \mathbf{I})(\mathbf{X} - \mathbf{C}) = 0$$

where the cone vertex is $\mathbf{C}$, the cone axis has direction $\mathbf{A}$, and the cone angle is θ (measured from the cone axis). Substituting the line equation $\mathbf{P} + t\mathbf{D}$ into the cone equation produces a quadratic function of t that may be solved in the same manner as the ones for spheres and ellipsoids.

For a single-sided cone, if T is a root of the quadratic function in t , you must verify that the corresponding point is on your half of the double-sided cone, $\delta = \mathbf{A} \cdot (T\mathbf{D} + \mathbf{P} - \mathbf{C}) \geq 0$. If the cone is also truncated by the plane $\mathbf{A} \cdot (\mathbf{X} - \mathbf{C}) = h$, where $h > 0$ is the truncated cone height, then you must additionally verify that $\delta \leq h$. Moreover, you must check for an intersection of the line with the truncating plane.

15.7 CULLING OBJECTS BY PLANES

One of the performance gains when drawing a scene graph is to eliminate subtrees of the scene that are outside the view frustum. This process is called object culling; see Section 2.4. Each node of the subtree has a bounding volume that is compared to each of the six frustum planes. If the bounding volume is outside any of these planes,

it is outside the frustum and, therefore, not visible. In this case, the subtree rooted at the node does not need to be traversed during the drawing pass.

Object culling may be viewed as a test-intersection query, with the minor modification that if the bounding volume does not intersect a culling plane, we want to know which side of the plane it is on. This section discusses algorithms for culling of planar components and various bounding volumes. All such components and bounding volumes are assumed to be convex.

In all cases, the strategy is simply to project the objects onto a normal line to the plane and determine if the projection set intersects the plane, lies completely on one side of the plane, or lies completely on the other side of the plane. The plane equation used in all sections is $\mathbf{N} \cdot \mathbf{X} = d$, where $\mathbf{N}$ is unit length and d is the plane constant. For culling purposes, the frustum is on the positive side of the plane, which is the side to which $\mathbf{N}$ points. The normal line for the projection is $t\mathbf{N}$, so the plane itself projects to $t = d$. Since the objects to be projected are convex, the projection must be a single interval $[t_0, t_1]$. The object is culled whenever $t_1 < d$. It suffices now to compute the projection interval for each object of interest.

15.7.1 ORIENTED BOXES

Let the oriented box have center $\mathbf{C}$, orthonormal axis directions $\mathbf{U}_i$, and extents e_i for $0 \leq i \leq 2$. The box is culled if all its vertices are outside the plane. The obvious algorithm of testing if all eight vertices are on the negative side of the plane requires eight comparisons of the form $\mathbf{N} \cdot \mathbf{P} < d$. The vertices are written in box coordinates as

$$\mathbf{P} = \mathbf{C} \pm e_0\mathbf{U}_0 \pm \sigma_1 e_1\mathbf{U}_1 \pm e_2\mathbf{U}_2$$

The projections are

$$\mathbf{N} \cdot \mathbf{P} = \mathbf{N} \cdot \mathbf{C} \pm e_0\mathbf{N} \cdot \mathbf{U}_0 \pm e_1\mathbf{N} \cdot \mathbf{U}_1 \pm e_2\mathbf{N} \cdot \mathbf{U}_2 \quad (15.1)$$

The four dot products are computed once, each dot product using three multiplications and two additions. Each test requires an additional three multiplications and four additions. The eight tests therefore require 76 operations. The projection interval is computed by selecting the minimum and maximum values for $\mathbf{N} \cdot \mathbf{P}$.

A faster algorithm is to determine the extreme points for the box in the direction of $\mathbf{N}$. Geometrically, the extreme points are corners if $\mathbf{N}$ is not parallel to any box axis; an entire edge when $\mathbf{N}$ is perpendicular to one axis but not to the other two axes; or an entire face when $\mathbf{N}$ is parallel to a box axis. In all cases, (at least) two vertices are extreme. Algebraically, the idea is to analyze the expression for $\mathbf{N} \cdot \mathbf{P}$. To make $\mathbf{N} \cdot \mathbf{P}$ as large as possible, in Equation (15.1) choose the signs on each of the three terms to make $x_i\mathbf{N} \cdot \mathbf{U}_i$ as large as possible. It is not actually necessary to write code to do this. What matters to us is the largest value, which must be the

absolute values of the terms. The same is true for the smallest value. The interval of projection is

$$[\mathbf{N} \cdot \mathbf{C} - r, \mathbf{N} \cdot \mathbf{C} + r], \quad r = e_0|\mathbf{N} \cdot \mathbf{U}_0| + e_1|\mathbf{N} \cdot \mathbf{U}_1| + e_2|\mathbf{N} \cdot \mathbf{U}_2|$$

Computing r requires four dot products, three multiplications, and three additions for a total operation count of 26, clearly less expensive than projecting all the vertices and selecting the extreme values. The pseudocode is

```
bool Culled (Box box, Plane plane)
{
    r = box.e[0] * |Dot(plane.N, box.U[0])| +
        box.e[1] * |Dot(plane.N, box.U[1])| +
        box.e[2] * |Dot(plane.N, box.U[2])|;

    return Dot(plane.N, box.C) + r < plane.d;
}
```

15.7.2 SPHERES

Let the sphere have center $\mathbf{C}$ and radius r . The extreme points of the sphere in the normal direction are $\mathbf{C} \pm r\mathbf{N}$. The projection of the sphere onto a normal line is

$$[\mathbf{N} \cdot \mathbf{C} - r, \mathbf{N} \cdot \mathbf{C} + r]$$

The pseudocode is

```
bool Culled (Sphere sphere, Plane plane)
{
    return Dot(plane.N, sphere.C) + sphere.r < plane.d;
}
```

15.7.3 CAPSULES

Let the capsule segment have center $\mathbf{C}$, direction $\mathbf{D}$, and extent e . Let the capsule radius be r . Since the capsule is a convex object, the minimum and maximum projected values are determined by the projections of the spheres at the endpoints of the segment. Each sphere has two extreme points in the normal direction, all four represented by

$$\mathbf{C} \pm e\mathbf{D} \pm r\mathbf{N}$$

The projection interval is

$$[\mathbf{N} \cdot \mathbf{C} - e|\mathbf{N} \cdot \mathbf{D}| - r, \mathbf{N} \cdot \mathbf{C} + e|\mathbf{N} \cdot \mathbf{D}| + r]$$

The pseudocode is

```
bool Culled (Capsule capsule, Plane plane)
{
    float r = capsule.r + capsule.e * |Dot(plane.N,capsule.D)|;
    return Dot(plane.N,capsule.C) + r < plane.d;
}
```

15.7.4 LOZENGES

Let the lozenge rectangle have center $\mathbf{C}$, directions $\mathbf{D}_0$ and $\mathbf{D}_1$, and extents e_0 and e_1 . Let the lozenge radius be r . Since the lozenge is a convex object, the minimum and maximum projected values are determined by the projections of the spheres at the corners of the rectangle. Each sphere has two extreme points in the normal direction, all eight represented by

$$\mathbf{C} \pm e_0\mathbf{D}_0 \pm e_1\mathbf{D}_1 \pm r\mathbf{N}$$

The projection interval is

$$[\mathbf{N} \cdot \mathbf{C} - e_0|\mathbf{N} \cdot \mathbf{D}_0| - e_1|\mathbf{N} \cdot \mathbf{D}_1| - r, \mathbf{N} \cdot \mathbf{C} + e_0|\mathbf{N} \cdot \mathbf{D}_0| + e_1|\mathbf{N} \cdot \mathbf{D}_1| + r]$$

The pseudocode is

```
bool Culled (Lozenge lozenge, Plane plane)
{
    float r = lozenge.r +
        lozenge.e[0] * |Dot(plane.N,lozenge.D[0])| +
        lozenge.e[1] * |Dot(plane.N,lozenge.D[1])|;

    return Dot(plane.N,lozenge.C) + r < plane.d;
}
```

15.7.5 ELLIPSOIDS

An ellipsoid is represented by the quadratic equation

$$Q(\mathbf{X}) = (\mathbf{X} - \mathbf{C})^T \mathbf{M} (\mathbf{X} - \mathbf{C}) = 1$$

where $\mathbf{C}$ is the center of the ellipsoid, $\mathbf{M}$ is a positive definite matrix, and $\mathbf{X}$ is any point on the ellipsoid. Figure 15.8 shows the projected interval for an ellipsoid but translated by subtracting the plane constant.

The projection interval is

$$[\mathbf{N} \cdot \mathbf{C} - r, \mathbf{N} \cdot \mathbf{C} + r]$$

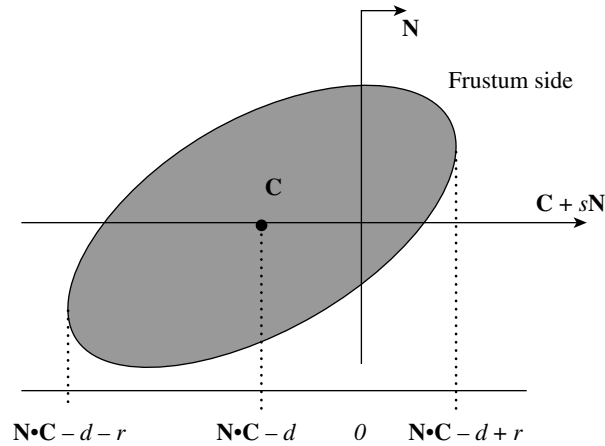


Figure 15.8 Projection of ellipsoid and frustum plane, no-cull case.

The construction of r is as follows. The points $\mathbf{X}$ that project to the endpoints of the interval must occur where the normals to the ellipsoid are parallel to $\mathbf{N}$. The gradient of $Q(\mathbf{X})$ is a normal direction for the point, $\nabla Q = 2M(\mathbf{X} - \mathbf{C})$. Thus, $\mathbf{X}$ must be a solution to $M(\mathbf{X} - \mathbf{C}) = \lambda\mathbf{N}$ for some scalar λ . Inverting M and multiplying yields $\mathbf{X} - \mathbf{C} = \lambda M^{-1}\mathbf{N}$. Replacing this in the quadratic equation yields

$$1 = \lambda^2 (M^{-1}\mathbf{N})^T M (M^{-1}\mathbf{N}) = \lambda^2 \mathbf{N}^T M^{-1} \mathbf{N}$$

Finally,

$$r = \mathbf{N} \cdot (\mathbf{X} - \mathbf{C}) = \lambda \mathbf{N}^T M^{-1} \mathbf{N}$$

so $r = \sqrt{\mathbf{N}^T M^{-1} \mathbf{N}}$. The pseudocode is

```
bool Culled (Ellipsoid ellipsoid, Plane plane)
{
    float NdCmD = Dot(plane.N, ellipsoid.C) - plane.d;
    if (NdCmD < 0)
    {
        float rSqr = Dot(plane.N, ellipsoid.Minverse * plane.N);
        return NdCmD * NdCmD >= rSqr;
    }
    return false;
}
```

The pseudocode is structured to compute r^2 to avoid the square root calculation and to avoid computing r^2 at all if the center of the ellipsoid is on the frustum side of the plane. The inverse of M is assumed to have been precomputed for use in such queries.

15.7.6 CYLINDERS

The finite cylinder has center $\mathbf{C}$, axis direction $\mathbf{D}$, height h , and radius ρ . Points in the cylinder are parameterized by

$$\mathbf{X}(t, \theta) = \mathbf{C} + t\mathbf{D} + \rho((\cos \theta)\mathbf{U} + (\sin \theta)\mathbf{V})$$

for $|t| \leq h/2$ and $\theta \in [0, 2\pi)$, and where $\{\mathbf{U}, \mathbf{V}, \mathbf{D}\}$ is an orthonormal set. The projections onto the normal line are

$$\mathbf{N} \cdot \mathbf{X}(t, \theta) = \mathbf{N} \cdot \mathbf{C} + t\mathbf{N} \cdot \mathbf{D} + \rho((\cos \theta)\mathbf{N} \cdot \mathbf{U} + (\sin \theta)\mathbf{N} \cdot \mathbf{V})$$

The projection interval is

$$[\mathbf{N} \cdot \mathbf{C} - r, \mathbf{N} \cdot \mathbf{C} + r]$$

where

$$r = (h/2)|\mathbf{N} \cdot \mathbf{D}| + \rho\sqrt{(\mathbf{N} \cdot \mathbf{U})^2 + (\mathbf{N} \cdot \mathbf{V})^2} = (h/2)|\mathbf{N} \cdot \mathbf{D}| + \rho\sqrt{1 - (\mathbf{N} \cdot \mathbf{D})^2}$$

The square root term appears because the choice for $(\cos \theta, \sin \theta)$ to maximize the dot product

$$(\cos \theta, \sin \theta) \cdot (\mathbf{N} \cdot \mathbf{U}, \mathbf{N} \cdot \mathbf{V})$$

is a unit-length vector in the direction of the other vector:

$$(\cos \theta, \sin \theta) = \frac{(\mathbf{N} \cdot \mathbf{U}, \mathbf{N} \cdot \mathbf{V})}{|(\mathbf{N} \cdot \mathbf{U}, \mathbf{N} \cdot \mathbf{V})|} = \sqrt{(\mathbf{N} \cdot \mathbf{U}, \mathbf{N} \cdot \mathbf{V})}$$

I have also used the identity $(\mathbf{N} \cdot \mathbf{U})^2 + (\mathbf{N} \cdot \mathbf{V})^2 + (\mathbf{N} \cdot \mathbf{D})^2 = 1$ since $\mathbf{N}$ is unit length in any orthonormal coordinate system. The pseudocode for the culling is

```
bool Culled (Cylinder cylinder, Plane plane)
{
    float NdD = Dot(plane.N, cylinder.D);
    float r = 0.5 * h * |NdD| + cylinder.r * sqrt(1 - NdD * NdD);
    return Dot(plane.N, cylinder.C) + r < plane.d;
}
```

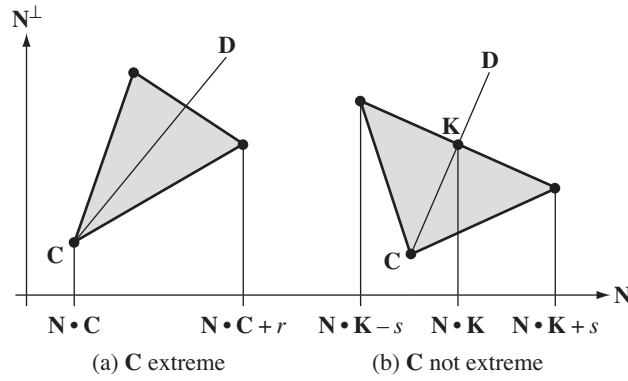


Figure 15.9 (a) The cone vertex (C) is an extreme point in the normal direction. (b) The cone vertex is not an extreme point in the normal direction.

15.7.7 CONES

The truncated cone has vertex C , axis direction D , angle θ , and height h . The truncating plane is $D \cdot (X - C) = h$. The radius of the end disk on the truncation plane is $\rho = h \tan \theta$. The cone points are parameterized by

$$X(t, \phi) = C + tD + (t \tan \theta)(\cos \phi U + \sin \phi V)$$

for $0 \leq t \leq h$ and $\phi \in [0, 2\pi)$, and where $\{U, V, D\}$ is an orthonormal set. The projections onto the normal line are

$$N \cdot X(t, \phi) = N \cdot C + tN \cdot D + (t \tan \theta)((\cos \phi)N \cdot U + (\sin \phi)N \cdot V)$$

The extreme values are slightly more complicated to extract. They depend on whether the cone vertex projects to one of the extrema. Figure 15.9 illustrates the possibilities. The extreme points of the cone depend on the angle that D forms with the unit-length vector $N^\perp$, which is a vector perpendicular to N and has the property $D \cdot N^\perp \geq 0$. In fact, some vector algebra shows that

$$D \cdot N^\perp = \sqrt{1 - (N \cdot D)^2}$$

The intersection of the cone axis and the cone disk cap is the point $K = C + hD$. The scalar values in the figure are

$$s = (h \tan \theta) \sqrt{1 - (N \cdot D)^2}, \quad r = h|N \cdot D| + s$$

The projection interval I is

$$I = \begin{cases} [\mathbf{N} \cdot \mathbf{C} - r, \mathbf{N} \cdot \mathbf{C}], & \mathbf{D} \cdot \mathbf{N}^\perp < \cos \theta, & \mathbf{D} \cdot \mathbf{N} < 0 \\ [\mathbf{N} \cdot \mathbf{C}, \mathbf{N} \cdot \mathbf{C} + r], & \mathbf{D} \cdot \mathbf{N}^\perp < \cos \theta, & \mathbf{D} \cdot \mathbf{N} > 0 \\ [\mathbf{N} \cdot \mathbf{K} - s, \mathbf{N} \cdot \mathbf{K} + s], & \mathbf{D} \cdot \mathbf{N}^\perp \geq \cos \theta & \end{cases}$$

The pseudocode for culling is

```
bool Culled (Cone cone, Plane plane)
{
    float NdC = Dot(plane.N, cone.C);
    float NdD = Dot(plane.N, cone.D);
    if (root >= cone.cosTheta && NdD < 0)
    {
        return NdC < plane.D;
    }
    float root = sqrt(1 - NdD * NdD);
    float r = h * NdD + cone.h * cone.tanTheta * root;
    return NdC + r < plane.d;
}
```

EXERCISE 15.4 Prove that

$$\mathbf{D} \cdot \mathbf{N}^\perp = \sqrt{1 - (\mathbf{N} \cdot \mathbf{D})^2}$$

Hint: Project out the $\mathbf{N}$ component from $\mathbf{D}$ and normalize the result to obtain $\mathbf{N}^\perp$. ■

15.7.8 CONVEX POLYGONS OR CONVEX POLYHEDRA

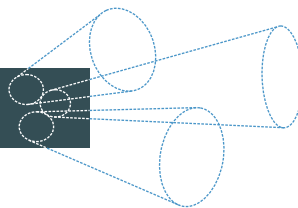
The projection interval for these objects may be computed by projecting the vertices and selecting the extreme values. The projection interval is

$$\left[\min_i \mathbf{N} \cdot \mathbf{P}_i, \max_i \mathbf{N} \cdot \mathbf{P}_i \right]$$

The object is culled whenever

$$\max_i \mathbf{N} \cdot \mathbf{P}_i < d$$

where d is the plane constant. If the convex polyhedron has a sufficiently large number of vertices, then the BSP-tree-based extremal query discussed in Section 8.1.1 may be used to compute the projection interval.



NUMERICAL METHODS

This chapter describes various numerical methods that are generally useful in computer graphics. Many of these are specifically useful in real-time game engines.

16.1 SYSTEMS OF EQUATIONS

The two types of systems that arise often in graphics applications are linear systems and polynomial systems. Linear systems are written in the form $A\mathbf{X} = \mathbf{b}$ for $n \times n$ matrix A and $n \times 1$ vectors $\mathbf{X}$ and $\mathbf{b}$. Both A and $\mathbf{b}$ are known. The unknowns are the components of $\mathbf{X}$. Polynomial systems are written in the form $p_i(\mathbf{X}) = 0$ for $0 \leq i < n$ for $n \times 1$ vector $\mathbf{X}$ and where p_i is a polynomial function.

16.1.1 LINEAR SYSTEMS

The standard approach to solving linear systems is Gaussian elimination with some type of pivoting. Standard numerical methods textbooks cover this topic in detail [BF01]. *Numerical Recipes in C* [PFTV88] also has good coverage. For more advanced topics on matrix systems, see [GL93].

For a 3×3 system, one symbolic method that is typically taught for solving the system uses the method of cofactors to invert A . If A is invertible, then $A^{-1} = A^{\text{Adj}} / \det(A)$, where A^{Adj} is the adjoint matrix, the transpose of the matrix of cofactors for A . The solution to the system is

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \frac{1}{\det(A)} \begin{bmatrix} a_{11}a_{22} - a_{12}a_{21} & a_{02}a_{21} - a_{01}a_{22} & a_{01}a_{12} - a_{02}a_{11} \\ a_{12}a_{20} - a_{10}a_{22} & a_{00}a_{22} - a_{02}a_{20} & a_{02}a_{10} - a_{00}a_{12} \\ a_{10}a_{21} - a_{11}a_{20} & a_{01}a_{20} - a_{00}a_{21} & a_{00}a_{11} - a_{01}a_{10} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

where

$$\det(A) = a_{00}(a_{11}a_{22} - a_{12}a_{21}) + a_{01}(a_{12}a_{20} - a_{10}a_{22}) + a_{02}(a_{10}a_{21} - a_{11}a_{20})$$

For $n \times n$ systems, the method of cofactors to invert A is $O(n!)$. Gaussian elimination is $O(n^3)$, so it is not difficult to see that Gaussian elimination is faster as n gets large. Moreover, the numerical stability of elimination algorithms is greatly desired. However, for $n = 3$, the asymptotic analysis is not particularly relevant. On most platforms, inversion of a 3×3 matrix is faster using cofactors than Gaussian elimination because a generic Gaussian elimination package requires some overhead, in particular loop iterations, which costs cycles. The speedup by using cofactors can be quite significant.

16.1.2 POLYNOMIAL SYSTEMS

Consider first the example of determining where two circles intersect in the plane. The equations for the circles are $(x - x_0)^2 + (y - y_0)^2 = r_0^2$ and $(x - x_1)^2 + (y - y_1)^2 = r_1^2$. The intersections (if any) are those (x, y) that solve both equations simultaneously. From the geometry there is either no solution (circles are disjoint), one solution (circles are tangent to each other), two solutions (circles interpenetrate), or infinitely many solutions (the circles are identical). If the circles are concentric, $x_0 = x_1$ and $y_0 = y_1$, then there is no intersection when $r_0 \neq r_1$ or infinitely many intersections when $r_0 = r_1$. Otherwise, suppose that either $x_0 \neq x_1$ or $y_0 \neq y_1$. The two quadratic equations can be solved by eliminating one of the variables. The second equation is subtracted from the first to obtain the linear equation

$$2(x_1 - x_0)x + x_0^2 - x_1^2 + 2(y_1 - y_0)y + y_0^2 - y_1^2 = r_0^2 - r_1^2$$

If $|y_1 - y_0| \geq |x_1 - x_0|$, solve for

$$y - y_0 = \frac{r_0^2 - r_1^2 + (x_1 - x_0)^2 + (y_0 - y_1)^2 - 2(x_1 - x_0)(x - x_0)}{2(y_1 - y_0)} = \frac{a(x - x_0) + b}{2(y_0 - y_1)}$$

Replace this in the first equation to obtain

$$(4(y_1 - y_0)^2 + a^2)(x - x_0)^2 + 2ab(x - x_0) + b^2 - 4(y_1 - y_0)^2 r_0^2 = 0$$

This is a quadratic equation in the single variable x and can be solved accordingly for up to two real-valued solutions. For each solution, the corresponding value of y is computed. The final pairs (x, y) must be tested for validity since extraneous solutions

might have been generated because of handling both signs on the square root in the quadratic formula.

The general problem of solving two quadratic equations in two unknowns is presented here. Given $P_0(x, y) = a_0x^2 + b_0xy + c_0y^2 + d_0x + e_0y + f_0$ and $P_1(x, y) = a_1x^2 + b_1xy + c_1y^2 + d_1x + e_1y + f_1$, find all solutions to $P_0(x, y) = 0$ and $P_1(x, y) = 0$. The solutions to $P_0(x, y) = 0$ and $P_1(x, y) = 0$ are found by elimination.

The two polynomials $f(x) = \alpha_0 + \alpha_1x + \alpha_2x^2$ and $g(x) = \beta_0 + \beta_1x + \beta_2x^2$ have a common root if and only if the *Bézout determinant* is zero:

$$(\alpha_2\beta_1 - \alpha_1\beta_2)(\alpha_1\beta_0 - \alpha_0\beta_1) - (\alpha_2\beta_0 - \alpha_0\beta_2)^2 = 0$$

and in which case the common root is

$$\bar{x} = (\alpha_2\beta_0 - \alpha_0\beta_2)/(\alpha_1\beta_2 - \alpha_2\beta_1)$$

The common root to $f(x) = 0$ and $g(x) = 0$ is obtained from the linear equation $\beta_2f(x) - \alpha_2g(x) = 0$. If the coefficient of x is zero, then f and g either have no common root or are the same polynomial (modulo a constant multiplier). Replacing the common root into $f(x) = 0$ yields the Bézout determinant.

The simultaneous quadratic equations are $P_i(x, y) = (a_i)x^2 + (b_iy + d_i)x + (c_iy^2 + e_iy + f_i)$ for $i = 0, 1$. The Bézout determinant is a quartic polynomial

$$R(y) = u_0 + u_1y + u_2y^2 + u_3y^3 + u_4y^4$$

where

$$u_0 = v_2v_{10} - v_4^2$$

$$u_1 = v_0v_{10} + v_2(v_7 + v_9) - 2v_3v_4$$

$$u_2 = v_0(v_7 + v_9) + v_2(v_6 - v_8) - v_3^2 - 2v_1v_4$$

$$u_3 = v_0(v_6 - v_8) + v_2v_5 - 2v_1v_3$$

$$u_4 = v_0v_5 - v_1^2$$

with

$$v_0 = a_0b_1 - a_1b_0 \quad v_4 = a_0f_1 - a_1f_0 \quad v_8 = c_0d_1 - c_1d_0$$

$$v_1 = a_0c_1 - a_1c_0 \quad v_5 = b_0c_1 - b_1c_0 \quad v_9 = d_0e_1 - d_1e_0$$

$$v_2 = a_0d_1 - a_1d_0 \quad v_6 = b_0e_1 - b_1e_0 \quad v_{10} = d_0f_1 - d_1f_0$$

$$v_3 = a_0e_1 - a_1e_0 \quad v_7 = b_0f_1 - b_1f_0$$

For each root $\bar{y}$ to $R(\bar{y}) = 0$, solve $P_0(x, \bar{y}) = 0$ for up to two values $\bar{x}$. Eliminate any extraneous solution $(\bar{x}, \bar{y})$ by verifying that $P_i(\bar{x}, \bar{y}) = 0$ for $i = 0, 1$.

[WG95a] and [WG95b] discuss in detail the general handling of polynomial systems using elimination and resultants. For three or more variables, the constructions can be quite complex.

16.2 EIGENSYSTEMS

Given an $n \times n$ matrix A , an *eigensystem* is of the form $AX = \lambda X$ or $(A - \lambda I)X = 0$. It is required that there be solutions $X \neq 0$. For this to happen, the matrix $A - \lambda I$ must be noninvertible. This is the case when $\det(A - \lambda I) = 0$, a polynomial in λ of degree n called the *characteristic polynomial* for A . For each root λ , the matrix $A - \lambda I$ is computed, and the system $(A - \lambda I)X = 0$ is solved for nonzero solutions. Although standard linear algebra textbooks show numerous examples for doing this symbolically, most applications require a robust numerical method for doing so. In particular, if $n \geq 5$, there are no closed formulas for roots to polynomials, so numerical methods must be applied. A good reference on solving eigensystems is [PFTV88]. An excellent reference for numerical methods relating to matrices is [GL93]. An excellent reference for matrix analysis is [HJ85].

Most of the applications in graphics that require eigensystems have symmetric matrices. The numerical methods are quite good for these since a full basis of eigenvectors is guaranteed. The standard approach is to apply orthogonal transformations, called *Householder transformations*, to reduce A to a tridiagonal matrix. The QR algorithm is applied iteratively to reduce the tridiagonal matrix to a diagonal one. [PFTV88] and [GL93] advise using a QL algorithm with implicit shifting to be as robust as possible.

For $n = 3$, the problem can be solved by simply computing the roots of $\det(A - \lambda I) = 0$. Often the numerical issues can be avoided since the end result is some visual presentation of data where the numerical error is not as important as for applications that require high precision, but generally you want to avoid the closed-form equations for roots because of their known nonrobustness problems.

16.2.1 EXTREMA OF QUADRATIC FORMS

Let A be an $n \times n$ symmetric matrix. The function $Q : \mathbb{R}^n \rightarrow \mathbb{R}$ defined by $Q(V) = V^T A V$ for $|V| = 1$ is called a *quadratic form*. Since Q is defined on the unit sphere in $\mathbb{R}^n$, a compact set, and since Q is continuous, it must have a maximum and a minimum on this set.

Let $V = \sum_{i=1}^n c_i V_i$, where $AV_i = \lambda_i V_i$, $\lambda_1 \leq \dots \leq \lambda_n$, and $\sum_{i=1}^n c_i^2 = 1$. That is, the λ_i are the eigenvalues of A , and the V_i are corresponding eigenvectors. Expanding the quadratic yields

$$Q(V) = \left(\sum_{i=1}^n c_i V_i^T \right) A \left(\sum_{j=1}^n c_j V_j \right) = \sum_{i=1}^n \sum_{j=1}^n c_i c_j V_i^T A V_j = \sum_{k=1}^n \lambda_k c_k^2$$

The rightmost summation is a convex combination of the eigenvalues of A , so its maximum is λ_n and occurs when $c_n = 1$ and all other $c_i = 0$. The point at which the maximum is attained is $\mathbf{V}_n$. Similarly, the minimum is λ_1 and occurs when $c_1 = 1$ and all other $c_i = 0$. The point at which the minimum is attained is $\mathbf{V}_1$.

16.2.2 EXTREMA OF CONSTRAINED QUADRATIC FORMS

In some applications it is desirable to find the extrema of a quadratic form defined on the unit hypersphere S^{n-1} , but restricted to the intersection of this hypersphere with a hyperplane $\mathbf{N} \cdot \mathbf{x} = 0$ for some special normal vector $\mathbf{N}$. Let A be an $n \times n$ symmetric matrix. Let $\mathbf{N} \in \mathbb{R}^n$ be a unit-length vector. Let $\{\mathbf{N}\}^\perp$ denote the orthogonal complement of $\mathbf{N}$; this is the largest-dimension vector space whose vectors are all perpendicular to $\mathbf{N}$. Define $Q : \{\mathbf{N}\}^\perp \rightarrow \mathbb{R}$ by $Q(\mathbf{V}) = \mathbf{V}^T \mathbf{A} \mathbf{V}$, where $|\mathbf{V}| = 1$. Now Q is defined on the unit sphere in the $(n-1)$ -dimensional space $\{\mathbf{N}\}^\perp$, so it must have a maximum and a minimum.

Let $\mathbf{V}_1$ through $\mathbf{V}_{n-1}$ be an orthonormal basis for $\{\mathbf{N}\}^\perp$. Let $\mathbf{V} = \sum_{i=1}^{n-1} c_i \mathbf{V}_i$, where $\sum_{i=1}^{n-1} c_i^2 = 1$. Let $\mathbf{A} \mathbf{V}_i = \sum_{j=1}^{n-1} \alpha_{ji} \mathbf{V}_j + \alpha_{ni} \mathbf{N}$, where $\alpha_{ji} = \mathbf{V}_j^T \mathbf{A} \mathbf{V}_i$ for $1 \leq i \leq n-1$ and $1 \leq j \leq n-1$, and where $\alpha_{ni} = \mathbf{N}^T \mathbf{A} \mathbf{V}_i$ for $1 \leq i \leq n-1$. Expanding the quadratic form yields

$$Q(\mathbf{V}) = \left(\sum_{i=1}^{n-1} c_i \mathbf{V}_i^T \right) A \left(\sum_{j=1}^{n-1} c_j \mathbf{V}_j \right) = \sum_{i=1}^{n-1} \sum_{j=1}^{n-1} c_i c_j \alpha_{ij} = \mathbf{C}^T \bar{\mathbf{A}} \mathbf{C} = P(\mathbf{C})$$

where quadratic form $P : \mathbb{R}^{n-1} \rightarrow \mathbb{R}$ satisfies the conditions for the maximization in the last section. Thus, $\max Q(\mathbf{V}) = \max P(\mathbf{C})$, which occurs for $\mathbf{C}$ and λ such that $\bar{\mathbf{A}} \mathbf{C} = \lambda \mathbf{C}$ and λ is the maximum eigenvalue of $\bar{\mathbf{A}}$. The following calculations lead to a matrix formulation for determining the maximum value:

$$\begin{aligned} \sum_{j=1}^{n-1} \alpha_{ij} c_j &= \lambda c_i \\ \sum_{j=1}^{n-1} c_j \mathbf{V}_i &= \lambda c_i \mathbf{V}_i \\ \sum_{i=1}^{n-1} \sum_{j=1}^{n-1} \alpha_{ij} c_j \mathbf{V}_i &= \lambda \sum_{i=1}^{n-1} c_i \mathbf{V}_i \\ \sum_{j=1}^{n-1} \left(\sum_{i=1}^{n-1} \alpha_{ij} \mathbf{V}_i \right) c_j &= \lambda \mathbf{V} \end{aligned}$$

$$\begin{aligned}
\sum_{j=1}^{n-1} (A\mathbf{V}_j - \alpha_{nj}\mathbf{N}) c_j &= \lambda \mathbf{V} \\
A \left(\sum_{j=1}^{n-1} c_j \mathbf{V}_j \right) - \left(\sum_{j=1}^{n-1} \alpha_{nj} c_j \right) \mathbf{N} &= \lambda \mathbf{V} \\
A\mathbf{V} - (\mathbf{N}^T A \mathbf{V}) \mathbf{N} &= \lambda \mathbf{V} \\
(I - \mathbf{N}\mathbf{N}^T)A\mathbf{V} &= \lambda \mathbf{V}
\end{aligned}$$

Therefore, $\max Q(\mathbf{V}) = \lambda_{n-1} = Q(\mathbf{V}_{n-1})$, where λ_{n-1} is the maximum eigenvalue corresponding to the eigenvector $\mathbf{V}_{n-1}$ of $(I - \mathbf{N}\mathbf{N}^T)A$. Note that $n - 1$ of the eigenvectors are in $\{\mathbf{N}\}^\perp$. The remaining eigenvector is $\mathbf{V}_n = A^{\text{Adj}}\mathbf{N}$, where $AA^{\text{Adj}} = (\det A)I$ and $\lambda_n = 0$.

16.3 LEAST-SQUARES FITTING

Least-squares fitting is the process of selecting a parameterized equation that represents a discrete set of points in a continuous manner. The parameters are estimated by minimizing a nonnegative function of the parameters. This section discusses fitting by lines, planes, quadratic curves, and quadric surfaces.

16.3.1 LINEAR FITTING OF POINTS $(x, f(x))$

This is the usual introduction to least-squares fit by a line when the data represents measurements where the y -component is assumed to be functionally dependent on the x -component. Given a set of samples $\{(x_i, y_i)\}_{i=1}^m$, determine A and B so that the line $y = Ax + B$ best fits the samples, in the sense that the sum of the squared errors between the y_i and the line values $Ax_i + B$ is minimized. Note that the error is measured only in the y -direction.

Define $E(A, B) = \sum_{i=1}^m [(Ax_i + B) - y_i]^2$. This function is nonnegative, and its graph is a paraboloid whose vertex occurs when the gradient satisfies $\nabla E = (0, 0)$. This leads to a system of two linear equations in A and B that can be easily solved:

$$(0, 0) = \nabla E = 2 \sum_{i=1}^m [(Ax_i + B) - y_i](x_i, 1)$$

and so

$$\begin{bmatrix} \sum_{i=1}^m x_i^2 & \sum_{i=1}^m x_i \\ \sum_{i=1}^m x_i & \sum_{i=1}^m 1 \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^m x_i y_i \\ \sum_{i=1}^m y_i \end{bmatrix}$$

The solution provides the least-squares solution $y = Ax + B$.

16.3.2 LINEAR FITTING OF POINTS USING ORTHOGONAL REGRESSION

It is also possible to fit a line using least-squares where the errors are measured *orthogonally* to the proposed line rather than measured vertically. The following argument holds for sample points and lines in n dimensions. Let the line be $\mathbf{L}(t) = t\mathbf{D} + \mathbf{A}$, where $\mathbf{D}$ is unit length. Define $\mathbf{X}_i$ to be the sample points; then

$$\mathbf{X}_i = \mathbf{A} + d_i\mathbf{D} + p_i\mathbf{D}_i^\perp$$

where $d_i = \mathbf{D} \cdot (\mathbf{X}_i - \mathbf{A})$ and $\mathbf{D}_i^\perp$ is some unit-length vector perpendicular to $\mathbf{D}$ with appropriate coefficient p_i . Define $\mathbf{Y}_i = \mathbf{X}_i - \mathbf{A}$. The vector from $\mathbf{X}_i$ to its projection onto the line is

$$\mathbf{Y}_i - d_i\mathbf{D} = p_i\mathbf{D}_i^\perp$$

The squared length of this vector is $p_i^2 = (\mathbf{Y}_i - d_i\mathbf{D})^2$. The error function for the least-squares minimization is $E(\mathbf{A}, \mathbf{D}) = \sum_{i=1}^m p_i^2$. Two alternative forms for this function are

$$E(\mathbf{A}, \mathbf{D}) = \sum_{i=1}^m \left(\mathbf{Y}_i^T R N \left[I - \mathbf{D}\mathbf{D}^T \right] \mathbf{Y}_i \right)$$

and

$$E(\mathbf{A}, \mathbf{D}) = \mathbf{D}^T \left(\sum_{i=1}^m \left[(\mathbf{Y}_i \cdot \mathbf{Y}_i) I - \mathbf{Y}_i \mathbf{Y}_i^T \right] \right) \mathbf{D} = \mathbf{D}^T M(\mathbf{A}) \mathbf{D}$$

Using the first form of E in the previous equation, take the derivative with respect to $\mathbf{A}$ to get

$$\frac{\partial E}{\partial \mathbf{A}} = -2 \left[I - \mathbf{D}\mathbf{D}^T \right] \sum_{i=1}^m \mathbf{Y}_i$$

This partial derivative is zero whenever $\sum_{i=1}^m \mathbf{Y}_i = 0$, in which case $\mathbf{A} = (1/m) \sum_{i=1}^m \mathbf{X}_i$, the average of the sample points.

Given $\mathbf{A}$, the matrix $M(\mathbf{A})$ is determined in the second form of the error function. The quantity $\mathbf{D}^T M(\mathbf{A}) \mathbf{D}$ is a quadratic form whose minimum is the smallest eigenvalue of $M(\mathbf{A})$. This can be found by standard eigensystem solvers. A corresponding unit-length eigenvector $\mathbf{D}$ completes our construction of the least-squares line.

For $n = 2$, if $\mathbf{A} = (a, b)$, then matrix $M(\mathbf{A})$ is given by

$$M(\mathbf{A}) = \left(\sum_{i=1}^m (x_i - a)^2 + \sum_{i=1}^n (y_i - b)^2 \right) \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} \sum_{i=1}^m (x_i - a)^2 & \sum_{i=1}^m (x_i - a)(y_i - b) \\ \sum_{i=1}^m (x_i - a)(y_i - b) & \sum_{i=1}^m (y_i - b)^2 \end{bmatrix}$$

For $n = 3$, if $\mathbf{A} = (a, b, c)$, then matrix $M(A)$ is given by

$$M(A) = \delta \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} \sum_{i=1}^m (x_i - a)^2 & \sum_{i=1}^m (x_i - a)(y_i - b) & \sum_{i=1}^m (x_i - a)(z_i - c) \\ \sum_{i=1}^m (x_i - a)(y_i - b) & \sum_{i=1}^m (y_i - b)^2 & \sum_{i=1}^m (y_i - b)(z_i - c) \\ \sum_{i=1}^m (x_i - a)(z_i - c) & \sum_{i=1}^m (y_i - b)(z_i - c) & \sum_{i=1}^m (z_i - c)^2 \end{bmatrix}$$

where

$$\delta = \sum_{i=1}^m (x_i - a)^2 + \sum_{i=1}^m (y_i - b)^2 + \sum_{i=1}^m (z_i - c)^2$$

16.3.3 PLANAR FITTING OF POINTS $(x, y, f(x, y))$

Here we assume that the z -component of the data is functionally dependent on the x - and y -components. Given a set of samples $\{(x_i, y_i, z_i)\}_{i=1}^m$, determine A , B , and C so that the plane $z = Ax + By + C$ best fits the samples in the sense that the sum of the squared errors between the z_i and the plane values $Ax_i + By_i + C$ is minimized. Note that the error is measured only in the z -direction.

Define $E(A, B, C) = \sum_{i=1}^m [(Ax_i + By_i + C) - z_i]^2$. This function is nonnegative, and its graph is a hyperparaboloid whose vertex occurs when the gradient satisfies $\nabla E = (0, 0, 0)$. This leads to a system of three linear equations in A , B , and C that can be easily solved:

$$(0, 0, 0) = \nabla E = 2 \sum_{i=1}^m [(Ax_i + By_i + C) - z_i](x_i, y_i, 1)$$

and so

$$\begin{bmatrix} \sum_{i=1}^m x_i^2 & \sum_{i=1}^m x_i y_i & \sum_{i=1}^m x_i \\ \sum_{i=1}^m x_i y_i & \sum_{i=1}^m y_i^2 & \sum_{i=1}^m y_i \\ \sum_{i=1}^m x_i & \sum_{i=1}^m y_i & \sum_{i=1}^m 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^m x_i z_i \\ \sum_{i=1}^m y_i z_i \\ \sum_{i=1}^m z_i \end{bmatrix}$$

The solution provides the least-squares solution $z = Ax + By + C$.

16.3.4 PLANAR FITTING OF POINTS USING ORTHOGONAL REGRESSION

It is also possible to fit a plane using least-squares where the errors are measured orthogonally to the proposed plane rather than measured vertically. The following argument holds for sample points and, actually, for hyperplanes in n dimensions.

Let the hyperplane be $\mathbf{N} \cdot (\mathbf{X} - \mathbf{A}) = 0$, where $\mathbf{N}$ is a unit-length normal to the hyperplane and $\mathbf{A}$ is a point on the hyperplane. Define $\mathbf{X}_i$ to be the sample points; then

$$\mathbf{X}_i = \mathbf{A} + \lambda_i \mathbf{N} + p_i \mathbf{N}_i^\perp$$

where $\lambda_i = \mathbf{N} \cdot (\mathbf{X}_i - \mathbf{A})$ and $\mathbf{N}_i^\perp$ is some unit-length vector perpendicular to $\mathbf{N}$ with appropriate coefficient p_i . Define $\mathbf{Y}_i = \mathbf{X}_i - \mathbf{A}$. The vector from $\mathbf{X}_i$ to its projection onto the hyperplane is $\lambda_i \mathbf{N}$. The squared length of this vector is $\lambda_i^2 = (\mathbf{N} \cdot \mathbf{Y}_i)^2$. The error function for the least-squares minimization is $E(\mathbf{A}, \mathbf{N}) = \sum_{i=1}^m \lambda_i^2$. Two alternative forms for this function are

$$E(\mathbf{A}, \mathbf{N}) = \sum_{i=1}^m \left(\mathbf{Y}_i^T [\mathbf{N} \mathbf{N}^T] \mathbf{Y}_i \right)$$

and

$$E(\mathbf{A}, \mathbf{N}) = \mathbf{N}^T \left(\sum_{i=1}^m \mathbf{Y}_i \mathbf{Y}_i^T \right) \mathbf{N} = \mathbf{N}^T M(\mathbf{A}) \mathbf{N}$$

Using the first form of E in the previous equation, take the derivative with respect to $\mathbf{A}$ to get

$$\frac{\partial E}{\partial \mathbf{A}} = -2 [\mathbf{N} \mathbf{N}^T] \sum_{i=1}^m \mathbf{Y}_i$$

This partial derivative is zero whenever $\sum_{i=1}^m \mathbf{Y}_i = 0$, in which case $\mathbf{A} = (1/m) \sum_{i=1}^m \mathbf{X}_i$ (the average of the sample points).

Given $\mathbf{A}$, the matrix $M(\mathbf{A})$ is determined in the second form of the error function. The quantity $\mathbf{N}^T M(\mathbf{A}) \mathbf{N}$ is a quadratic form whose minimum is the smallest eigenvalue of $M(\mathbf{A})$. This can be found by standard eigensystem solvers. A corresponding unit-length eigenvector $\mathbf{N}$ completes our construction of the least-squares hyperplane.

For $n = 3$, if $\mathbf{A} = (a, b, c)$, then matrix $M(\mathbf{A})$ is given by

$$M(\mathbf{A}) = \begin{bmatrix} \sum_{i=1}^m (x_i - a)^2 & \sum_{i=1}^m (x_i - a)(y_i - b) & \sum_{i=1}^m (x_i - a)(z_i - c) \\ \sum_{i=1}^m (x_i - a)(y_i - b) & \sum_{i=1}^m (y_i - b)^2 & \sum_{i=1}^m (y_i - b)(z_i - c) \\ \sum_{i=1}^m (x_i - a)(z_i - c) & \sum_{i=1}^m (y_i - b)(z_i - c) & \sum_{i=1}^m (z_i - c)^2 \end{bmatrix}$$

16.3.5 FITTING A CIRCLE TO 2D POINTS

Given a set of points $\{(x_i, y_i)\}_{i=1}^m$, $m \geq 3$, fit them with a circle $(x - a)^2 + (y - b)^2 = r^2$, where (a, b) is the circle center and r is the circle radius. An assumption

of this algorithm is that not all the points are collinear. The error function to be minimized is

$$E(a, b, r) = \sum_{i=1}^m (L_i - r)^2$$

where $L_i = \sqrt{(x_i - a)^2 + (y_i - b)^2}$. Take the partial derivative with respect to r to obtain

$$\frac{\partial E}{\partial r} = -2 \sum_{i=1}^m (L_i - r)$$

Setting equal to zero yields

$$r = \frac{1}{m} \sum_{i=1}^m L_i$$

Take the partial derivative with respect to a to obtain

$$\frac{\partial E}{\partial a} = 2 \sum_{i=1}^m (L_i - r) \frac{\partial L_i}{\partial a} = -2 \sum_{i=1}^m \left((x_i - a) + r \frac{\partial L_i}{\partial a} \right)$$

and take the partial derivative with respect to b to obtain

$$\frac{\partial E}{\partial b} = 2 \sum_{i=1}^m (L_i - r) \frac{\partial L_i}{\partial b} = -2 \sum_{i=1}^m \left((y_i - b) + r \frac{\partial L_i}{\partial b} \right)$$

Setting these two derivatives equal to zero yields

$$a = \frac{1}{m} \sum_{i=1}^m x_i + r \frac{1}{m} \sum_{i=1}^m \frac{\partial L_i}{\partial a}$$

and

$$b = \frac{1}{m} \sum_{i=1}^m y_i + r \frac{1}{m} \sum_{i=1}^m \frac{\partial L_i}{\partial b}$$

Replacing r by its equivalent from $\partial E / \partial r = 0$ and using $\partial L_i / \partial a = (a - x_i) / L_i$ and $\partial L_i / \partial b = (b - y_i) / L_i$ leads to two nonlinear equations in a and b :

$$a = \bar{x} + \bar{L} \bar{L}_a =: F(a, b)$$

$$b = \bar{y} + \bar{L} \bar{L}_b =: G(a, b)$$

where

$$\begin{aligned}\bar{x} &= \frac{1}{m} \sum_{i=1}^m x_i, & \bar{y} &= \frac{1}{m} \sum_{i=1}^m y_i, \\ \bar{L} &= \frac{1}{m} \sum_{i=1}^m L_i, & \bar{L}_a &= \frac{1}{m} \sum_{i=1}^m \frac{a - x_i}{L_i}, & \bar{L}_b &= \frac{1}{m} \sum_{i=1}^m \frac{b - y_i}{L_i}\end{aligned}$$

Fixed-point iteration can be applied to solving these equations: $a_0 = \bar{x}$, $b_0 = \bar{y}$, and $a_{i+1} = F(a_i, b_i)$ and $b_{i+1} = G(a_i, b_i)$ for $i \geq 0$.

16.3.6 FITTING A SPHERE TO 3D POINTS

Given a set of points $\{(x_i, y_i, z_i)\}_{i=1}^m$, $m \geq 4$, fit them with a sphere $(x - a)^2 + (y - b)^2 + (z - c)^2 = r^2$, where (a, b, c) is the sphere center and r is the sphere radius. An assumption of this algorithm is that not all the points are coplanar. The error function to be minimized is

$$E(a, b, c, r) = \sum_{i=1}^m (L_i - r)^2$$

where $L_i = \sqrt{(x_i - a)^2 + (y_i - b)^2 + (z_i - c)^2}$. Take the partial derivative with respect to r to obtain

$$\frac{\partial E}{\partial r} = -2 \sum_{i=1}^m (L_i - r)$$

Setting equal to zero yields

$$r = \frac{1}{m} \sum_{i=1}^m L_i$$

Take the partial derivative with respect to a to obtain

$$\frac{\partial E}{\partial a} = 2 \sum_{i=1}^m (L_i - r) \frac{\partial L_i}{\partial a} = -2 \sum_{i=1}^m \left((x_i - a) + r \frac{\partial L_i}{\partial a} \right)$$

take the partial derivative with respect to b to obtain

$$\frac{\partial E}{\partial b} = 2 \sum_{i=1}^m (L_i - r) \frac{\partial L_i}{\partial b} = -2 \sum_{i=1}^m \left((y_i - b) + r \frac{\partial L_i}{\partial b} \right)$$

and take the partial derivative with respect to c to obtain

$$\frac{\partial E}{\partial c} = 2 \sum_{i=1}^m (L_i - r) \frac{\partial L_i}{\partial c} = -2 \sum_{i=1}^m \left((z_i - c) + r \frac{\partial L_i}{\partial c} \right)$$

Setting these three derivatives equal to zero yields

$$a = \frac{1}{m} \sum_{i=1}^m x_i + r \frac{1}{m} \sum_{i=1}^m \frac{\partial L_i}{\partial a}$$

and

$$b = \frac{1}{m} \sum_{i=1}^m y_i + r \frac{1}{m} \sum_{i=1}^m \frac{\partial L_i}{\partial b}$$

and

$$c = \frac{1}{m} \sum_{i=1}^m z_i + r \frac{1}{m} \sum_{i=1}^m \frac{\partial L_i}{\partial c}$$

Replacing r by its equivalent from $\partial E / \partial r = 0$ and using $\partial L_i / \partial a = (a - x_i) / L_i$, $\partial L_i / \partial b = (b - y_i) / L_i$, and $\partial L_i / \partial c = (c - z_i) / L_i$ leads to three nonlinear equations in a , b , and c :

$$a = \bar{x} + \bar{L} \bar{L}_a =: F(a, b, c)$$

$$b = \bar{y} + \bar{L} \bar{L}_b =: G(a, b, c)$$

$$c = \bar{z} + \bar{L} \bar{L}_c =: H(a, b, c)$$

where

$$\bar{x} = \frac{1}{m} \sum_{i=1}^m x_i$$

$$\bar{y} = \frac{1}{m} \sum_{i=1}^m y_i$$

$$\bar{z} = \frac{1}{m} \sum_{i=1}^m z_i$$

$$\bar{L} = \frac{1}{m} \sum_{i=1}^m L_i$$

$$\bar{L}_a = \frac{1}{m} \sum_{i=1}^m \frac{a - x_i}{L_i}$$

$$\bar{L}_b = \frac{1}{m} \sum_{i=1}^m \frac{b - y_i}{L_i}$$

$$\bar{L}_c = \frac{1}{m} \sum_{i=1}^m \frac{c - z_i}{L_i}$$

Fixed-point iteration can be applied to solving these equations: $a_0 = \bar{x}$, $b_0 = \bar{y}$, $c_0 = \bar{z}$, and $a_{i+1} = F(a_i, b_i, c_i)$, $b_{i+1} = G(a_i, b_i, c_i)$, and $c_{i+1} = H(a_i, b_i, c_i)$ for $i \geq 0$.

16.3.7 FITTING A QUADRATIC CURVE TO 2D POINTS

Given a set of points $\{(x_i, y_i)\}_{i=0}^n$, a quadratic curve of the form $Q(x, y) = c_0 + c_1x + c_2y + c_3x^2 + c_4y^2 + c_5xy = 0$ is sought to fit the points. Given values c_i that provide the fit, any scalar multiple provides the same fit. To eliminate this degree of freedom, require that $\mathbf{C} = (c_0, \dots, c_5)$ have unit length. Define the vector variable $\mathbf{V} = (1, x, y, x^2, y^2, xy)$. The quadratic equation is restated as $Q(\mathbf{V}) = \mathbf{C} \cdot \mathbf{V} = 0$ and is a linear equation in the space of $\mathbf{V}$. Define $\mathbf{V}_i = (1, x_i, y_i, x_i^2, y_i^2, x_i y_i)$ for the i th data point. While generally $Q(\mathbf{V}_i)$ is not zero, the idea is to minimize the sum of squares

$$E(\mathbf{C}) = \left(\sum_{i=0}^n \mathbf{C} \cdot \mathbf{V}_i \right)^2 = \mathbf{C}^T \mathbf{M} \mathbf{C}$$

where $\mathbf{M} = \sum_{i=0}^n \mathbf{V}_i \mathbf{V}_i^T$ and subject to the constraint $|\mathbf{C}| = 1$. Now the problem is in the standard format for minimizing a quadratic form (see Section 16.2.1). The minimum value is the smallest eigenvalue of $\mathbf{M}$, and $\mathbf{C}$ is a corresponding unit-length eigenvector. The minimum itself can be used as a measure of how good the fit is (0 means the fit is exact).

If there is reason to believe the input points are nearly circular, a minor modification can be used in the construction. The circle is of the form $Q(x, y) = c_0 + c_1x + c_2y + c_3(x^2 + y^2) = 0$. The same construction can be applied where $\mathbf{V} = (1, x, y, x^2 + y^2)$ and $E(\mathbf{C}) = \mathbf{C}^T \mathbf{M} \mathbf{C}$ subject to $|\mathbf{C}| = 1$.

16.3.8 FITTING A QUADRIC SURFACE TO 3D POINTS

Given a set of points $\{(x_i, y_i, z_i)\}_{i=0}^n$, a quadric surface of the form $Q(x, y, z) = c_0 + c_1x + c_2y + c_3z + c_4x^2 + c_5y^2 + c_6z^2 + c_7xy + c_8xz + c_9yz = 0$ is sought to fit the points. Just like in the previous section, $\mathbf{C} = (c_i)$ is required to be unit length and

$\mathbf{V} = (1, x, y, z, x^2, y^2, z^2, xy, xz, yz)$. The quadratic form to minimize is $E(\mathbf{C}) = \mathbf{C}^T \mathbf{M} \mathbf{C}$, where $\mathbf{M} = \sum_{i=0}^2 \mathbf{V}_i \mathbf{V}_i^T$. The minimum value is the smallest eigenvalue of $\mathbf{M}$, and $\mathbf{C}$ is a corresponding unit-length eigenvector. The minimum itself can be used as a measure of how good the fit is (0 means the fit is exact).

If there is reason to believe the input points are nearly spherical, a minor modification can be used in the construction. The sphere is of the form $Q(x, y, z) = c_0 + c_1x + c_2y + c_3z + c_4(x^2 + y^2 + z^2) = 0$. The same construction can be applied where $\mathbf{V} = (1, x, y, z, x^2 + y^2 + z^2)$ and $E(\mathbf{C}) = \mathbf{C}^T \mathbf{M} \mathbf{C}$ subject to $|\mathbf{C}| = 1$.

16.4 MINIMIZATION

The generic problem is to find a global minimum for a function $f : D \subset \mathbb{R}^n \rightarrow \mathbb{R}$. The function is constrained to be at least continuous, and D is assumed to be a compact set. If the function is continuously differentiable, this fact can help in locating a minimum, but there are methods that do not require derivatives in finding one.

16.4.1 METHODS IN ONE DIMENSION

Consider $f : [t_{\min}, t_{\max}] \rightarrow \mathbb{R}$. If f is differentiable, then the global minimum must occur either at a point where $f' = 0$ or at one of the endpoints. The squared-distance function is quadratic and is defined on a compact interval. The minimum of that function occurs at an interior point of the interval, in which case the closest point is interior to the line segment or at an endpoint. Solving the problem $f'(t) = 0$ may be complicated in itself. This root-finding problem is described in Section 16.5.1.

Brent's Method

Continuous functions that are not necessarily differentiable must attain a minimum on a compact interval. A method to find the minimum that does not require derivatives or determining where the derivative is zero when the function is differentiable is very desirable. One such method, *Brent's method*, uses inverse parabolic interpolation in an iterative fashion.

The idea is to *bracket* the minimum by three points $(t_0, f(t_0))$, $(t_m, f(t_m))$, and $(t_1, f(t_1))$ for $t_{\min} \leq t_0 < t_m < t_1 \leq t_{\max}$, where $f(t_m) < f(t_0)$ and $f(t_m) < f(t_1)$. This means the function must decrease for some values of $t \in [t_0, t_m]$ and must increase for some values of $t \in [t_m, t_1]$, which guarantees that f has a local minimum somewhere in $[t_0, t_1]$. Brent's method attempts to narrow in on the local minimum, much like the bisection method narrows in on the root of a function (see Section 16.5).

The following is a variation of the description of Brent's method by [PFTV88]. The three bracketing points are fit with a parabola, $p(t)$. The vertex of the parabola is guaranteed to lie within (t_0, t_1) . Let $f_0 = f(t_0)$, $f_m = f(t_m)$, and $f_1 = f(t_1)$. The vertex of the parabola occurs at $t_v \in (t_0, t_1)$ and can be shown to be

$$t_v = t_m - \frac{1}{2} \frac{(t_1 - t_0)^2(f_0 - f_m) - (t_0 - t_m)^2(f_1 - f_m)}{(t_1 - t_m)(f_0 - f_m) - (t_0 - t_m)(f_1 - f_m)}$$

The function is evaluated there, $f_v = f(t_v)$. If $t_v < t_m$, then the new bracket is (t_0, f_0) , (t_v, f_v) , and (t_m, f_m) . If $t_v > t_m$, then the new bracket is (t_m, f_m) , (t_v, f_v) , and (t_1, f_1) . If $t_v = t_m$, the bracket cannot be updated in a simple way. Moreover, it is not sufficient to terminate the iteration here, because it is simple to construct an example where the three samples form an isosceles triangle whose vertex on the axis of symmetry is the parabola vertex, but the global minimum is far away from that vertex. One simple heuristic is to use the midpoint of one of the half-intervals, say, $t_b = (t_0 + t_m)/2$, evaluate $f_b = f(t_b)$, and compare to f_m . If $f_b > f_m$, then the new bracket is (t_b, f_b) , (t_m, f_m) , and (t_1, f_1) . If $f_b < f_m$, then the new bracket is (t_0, f_0) , (t_b, f_b) , and (t_m, f_m) . If $f_b = f_m$, the other half-interval can be bisected and the same tests repeated. If that also produces the pathological equality case, try a random sample from $[t_0, t_1]$. Once the new bracket is known, the method can be repeated until some stopping criterion is met.

Brent's method can also be modified to support derivative information [PFTV88].

16.4.2 METHODS IN MANY DIMENSIONS

Consider $f : D \subset \mathbb{R}^n \rightarrow \mathbb{R}$, where D is a compact set. Typically in graphics applications, D is a polyhedron or even a Cartesian product of intervals. If f is differentiable, then the global minimum must occur either at a point where $\nabla f = \mathbf{0}$ or on the boundary of D . In the latter case, if D is a polyhedron, then the restriction of f to each face of D produces the same type of minimization problem, but in one less dimension. Solving $\nabla f = \mathbf{0}$ is a root-finding problem and itself may be a difficult problem to solve.

Steepest Descent Search

Steepest descent search is a simple approach to searching for a minimum of a differentiable function. From calculus it is known that the direction in which f has its greatest rate of decrease is $-\nabla f$. Given an initial guess $\mathbf{X}$ for the minimum point, the function $\phi(t) = f(\mathbf{X} - t\nabla f(\mathbf{X}))$ is minimized using a 1D algorithm. If t' is the parameter at which the minimum occurs, then $\mathbf{X} \leftarrow \mathbf{X} - t'\nabla f(\mathbf{X})$, and the algorithm is repeated until a stopping condition is met. The condition is typically a measure of how different the last starting position is from the newly computed position.

The problem with this method is that it can be very slow. The pathological case is the minimization of a paraboloid $f(x, y) = (x/a)^2 + y^2$, where a is a very large number. The level sets are ellipses that are very elongated in the x -direction. For points not on the x -axis, the negative of the gradient vector tends to be nearly parallel to the y -axis. The search path will zig-zag back and forth across the x -axis, taking its time getting to the origin, where the global minimum occurs. A better approach is not to use the gradient vector, but to use the *conjugate direction*. For the paraboloid, no matter where the initial guess is, only two iterations using conjugate directions will always end up at the origin. These directions in a sense encode shape information about the level curves of the function.

Conjugate Gradient Search

This method attempts to choose a better set of directions than steepest descent for a minimization search. Only a brief summary is given here (for more details, see [PFTV88]). Two sequences of directions are built, a sequence of gradient directions $\mathbf{g}_i$ and a sequence of conjugate directions $\mathbf{h}_i$. The 1D minimizations are along lines corresponding to the conjugate directions. The following pseudocode uses the Polak and Ribiere formulation as mentioned in [PFTV88]. The function to be minimized is $E(\mathbf{X})$. The function `Minimize0n` minimizes the function along the line using a 1D minimizer. It returns the location $\mathbf{x}$ of the minimum and the function value `fval` at that minimum.

```
x = initial guess;
g = -gradient(E)(x);
h = g;
while (not done)
{
    line.origin = x;
    line.direction = h;
    Minimize0n(line,x,fval);
    if (stopping condition met)
    {
        return <x,fval>;
    }

    gNext = -gradient(E)(x);
    c = Dot(gNext-g,gNext)/Dot(g,g);
    g = gNext;
    h = g + c * h;
}
```

The stopping condition can be based on consecutive values of $fval$ and/or on consecutive values of x . The condition in [PFTV88] is based on consecutive function values, f_0 and f_1 , and a small tolerance value $\tau > 0$,

$$2|f_1 - f_0| \leq \tau(|f_0| + |f_1| + \epsilon)$$

for a small value $\epsilon > 0$ that supports the case when the function minimum is zero.

Powell's Direction Set Method

If f is continuous but not differentiable, then it attains a minimum on D . The search for the minimum simply cannot use derivative information. A method to find a minimum that does not require derivatives is *Powell's direction set method*. This method solves minimization problems along linear paths in the domain. The current candidate for the point at which the minimum occurs is updated to the minimum point on the current line under consideration. The next line is chosen to contain the current point and has a direction selected from a maintained set of direction vectors. Once all the directions have been processed, a new set of directions is computed. This is typically all but one of the previous set, but with the first direction removed and a new direction set to the current position minus the old position before the line minimizations were processed. The minimizations along the lines use something such as Brent's method since f restricted to the line is a 1D function. The fact that D is compact guarantees that the intersection of a line with D is a compact set. Moreover, if D is convex (and in most applications it is), then the intersection is a connected interval so that Brent's method can be applied to that interval (rather than applying it to each connected component of the intersection of a line with D). The pseudocode for Powell's method is

```
// F(x) is the function to be minimized.
n = dimension of domain;
directionSet = {d[0], ..., d[n - 1]}; // usually the standard axis
                                     // directions
x = xInitial = initial guess for minimum point;
while (not done)
{
    for (i = 0; i < n; i++)
    {
        line.origin = x;
        line.direction = d[i];
        MinimizeOn(line, x, fval);
    }

    conjugateDirection = x - xInitial;
```

```

    if (Length(conjugateDirection) is small)
    {
        return <x,fval>; // minimum found
    }

    for (i = 0; i <= n - 2; i++)
    {
        d[i] = d[i + 1];
    }
    d[n - 1] = conjugateDirection;
}

```

The function `Minimize0n` is the same one mentioned in the previous section on the conjugate gradient search.

16.5 ROOT FINDING

Given a continuous function $F : D \subset \mathbb{R}^n \rightarrow \mathbb{R}^n$, the problem is to find an $\mathbf{X}$ (or find a set of points) for which $F(\mathbf{X}) = 0$.

16.5.1 METHODS IN ONE DIMENSION

Given a continuous function $f : [a, b] \rightarrow \mathbb{R}$, the first question is whether or not $f(r) = 0$ for some $r \in [a, b]$. If $f(a)f(b) < 0$, then there is at least one root. However, there may be multiple roots. If a root r is computed, other analyses are required to locate other roots. For example, if f is a polynomial and r is a root, the function can be factored as $f(t) = (t - r)^p g(t)$, where $p \geq 1$ and g is a polynomial with $\text{degree}(g) = \text{degree}(f) - p$. The root-finding process is now continued with function g on $[a, b]$.

If $f(a)f(b) > 0$, there is no guarantee that f has a root on $[a, b]$. For problems of this type, a root-bounding preprocessing step can be used. The interval is partitioned into $t_i = a + i(b - a)/n$ for $0 \leq i \leq n$. If $f(t_i)f(t_{i+1}) < 0$ for some i , then that subinterval is bisected to locate a root. A reasonable choice of n will be related to what information the application knows about its function f .

Finally, it might be necessary to find roots of $f : \mathbb{R} \rightarrow \mathbb{R}$, where the domain of f is not a bounded interval. In this case, roots of f can be sought in the interval $[-1, 1]$. For $|t| \geq 1$, the function $g(t) = f(1/t)$ is defined for $t \in [-1, 1]$. Roots for g are sought on $[-1, 1]$. If $g(r) = 0$, then $f(1/r) = 0$.

Bisection

Bisection is the process of finding a root to a continuous function $f : [a, b] \rightarrow \mathbb{R}$ by bracketing a root with an interval, then successively bisecting the interval to narrow in on the root. Suppose that initially $f(a)f(b) < 0$. Since f is continuous, there must be a root $r \in (a, b)$. The midpoint of the interval is $m = (a + b)/2$. The function value $f(m)$ is computed and compared to the function values at the endpoints. If $f(a)f(m) < 0$, then the subinterval (a, m) brackets a root and the bisection process is repeated on that subinterval. If $f(m)f(b) < 0$, the subinterval (m, b) brackets a root and the bisection process is repeated instead on that subinterval. If $f(m) = 0$ or is zero within a specified tolerance, the process terminates. A stopping condition might also be based on the length of the current subinterval—that is, if the length becomes small enough, terminate the algorithm. If a root exists on $[a, b]$, bisection is guaranteed to find it. However, the rate of convergence is slow.

Newton's Method

Given a differentiable function $f : \mathbb{R} \rightarrow \mathbb{R}$, an initial guess is chosen about where f is zero, $(x_0, f(x_0))$. The tangent line to the graph at this point is used to update the estimate to a (hopefully) better one. The tangent line is $y - f(x_0) = f'(x_0)(x - x_0)$ and intersects the x -axis at $(0, x_1)$, so $-f(x_0) = f'(x_0)(x_1 - x_0)$. Assuming $f'(x_0) \neq 0$, solving for x_1 yields

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

The next point in the iteration is $(x_1, f(x_1))$ and the process is repeated until a stopping condition is met, typically one based on closeness of the function value to zero. Unlike bisection, the iterations are not guaranteed to converge, but if there is convergence, it is at a faster rate. Success depends a lot on the initial guess for x_0 .

Polynomial Roots

A polynomial of degree n is $f(t) = \sum_{i=0}^n a_i t^i$, where $a_n \neq 0$. While standard root finders may be applied to polynomials, a better approach takes advantage of the nature of such functions. For $2 \leq n \leq 4$, there are closed-form equations for the roots of the polynomial. Direct application of the formulas is possible, but numerical problems tend to occur, particularly when the polynomial has a root of multiplicity larger than 1. For example, the roots of a quadratic $f(t) = at^2 + bt + c$ are $t = (-b \pm \sqrt{b^2 - 4ac})/(2a)$. If $b^2 - 4ac = 0$, the quadratic has a double root $t = -b/(2a)$. However, numerical round-off errors might cause $b^2 - 4ac = -\epsilon < 0$ for very small ϵ . Another condition that leads to numerical problems is if a is nearly

zero. If so, it is possible to solve $g(t) = t^2 f(1/t) = ct^2 + bt + a = 0$ and get $t = (-b \pm \sqrt{b^2 - 4ac})/(2c)$. But the problem still exists if c is also nearly zero. Similar problems occur with the formulas for cubic and quartic polynomials.

An approach based on iteration schemes is to attempt to bracket the roots in a way that each bracketing interval contains exactly one root. For each such interval, bisection can be applied to find the root. A hybrid scheme is also possible that mixes bisection steps with Newton steps; the bisection step is used only when the Newton step generates an iterate outside the current bracketing interval. The hope is that the Newton iterates converge quickly to the root, but if they appear not to, bisection attempts to generate better initial guesses for the Newton iteration.

Bounding Roots by Derivative Sequences

A simple approach to the bracketing problem is to partition $\mathbb{R}$ into intervals, with the polynomial $f(t)$ monotone on each interval. If it can be determined where the derivative of the polynomial is zero, this set provides the partition. If d_i and d_{i+1} are consecutive values for which $f'(d_i) = f'(d_{i+1}) = 0$, then either $f'(t) > 0$ on (d_i, d_{i+1}) or $f'(t) < 0$ on (d_i, d_{i+1}) . In either case, f can have at most one root on the interval. The existence of this root is guaranteed by the condition $f(d_i)f(d_{i+1}) < 0$ or $f(d_i) = 0$ or $f(d_{i+1}) = 0$.

Solving $f'(t) = 0$ requires the same techniques as solving $f(t) = 0$. The difference is that $\text{degree}(f') = \text{degree}(f) - 1$. A recursive implementation is warranted for this problem; the base case is the constant polynomial that is either never zero or identically zero on the real line.

If $f'(t) \neq 0$ for $t \in (-\infty, d_0]$, it is possible that f has a root on the semi-infinite interval $(-\infty, d_0]$. Bisection does not help locate a root, because the interval is unbounded. However, it is possible to determine the largest bounded interval that contains the roots of a polynomial. The construction relies on the concepts of *spectral radius* and *norm of a matrix* [HJ85]. Given a square matrix A , the spectral radius, denoted $\rho(A)$, is the maximum of the absolute values of the eigenvalues for the matrix. A matrix norm of A , denoted $\|A\|$, is a scalar-valued function that must satisfy the five conditions: $\|A\| \geq 0$, $\|A\| = 0$ if and only if $A = 0$, $\|cA\| = |c|\|A\|$ for any scalar c , $\|A + B\| \leq \|A\| + \|B\|$, and $\|AB\| \leq \|A\|\|B\|$. The relationship between the spectral radius and any matrix norm is $\rho(A) \leq \|A\|$. Given $f(t) = \sum_{i=0}^n a_i t^i$, where $a_n = 1$, the *companion matrix* is

$$A = \begin{bmatrix} -a_{n-1} & -a_{n-2} & \cdots & -a_1 & -a_0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

The characteristic polynomial is $f(t) = \det(A - tI)$, so the roots of f are the eigenvalues of A . The spectral norm therefore provides a bound for the roots. Since

there are lots of matrix norms to choose from, there are many possible bounds. One such bound is Cauchy's bound,

$$|t| \leq \max\{|a_0|, 1 + |a_1|, \dots, 1 + |a_{n-1}|\} = 1 + \max\{|a_0|, \dots, |a_{n-1}|\}$$

Another bound that can be obtained is the Carmichael and Mason bound,

$$|t| \leq \sqrt{1 + \sum_{i=0}^{n-1} |a_i|^2}$$

If $a_0 \neq 0$, then $f(0) \neq 0$, so the roots of f are bounded away from zero. It is possible to construct lower bounds by using $g(t) = [t^n f(1/t)]/a_0$. The roots of $g(t)$ are the reciprocal roots of $f(t)$. Cauchy's bound applied to $g(t)$, then taking reciprocals, is

$$|t| \geq \frac{|a_0|}{1 + \max\{1, |a_1|, \dots, |a_{n-1}|\}}$$

The Carmichael and Mason bound is

$$|t| \geq \frac{|a_0|}{\sqrt{1 + \sum_{i=0}^{n-1} |a_i|^2}}$$

These bounds are used in the recursive call to determine where $f(t)$ is monotone. The polynomial can be factored $f(t) = t^p g(t)$, where $p \geq 0$ and g is a polynomial for which $g(0) \neq 0$. If $p = 0$, then $f = g$, and f is processed for $0 < a \leq |t| \leq b$, where a and b are bounds computed from the previously mentioned inequalities. If $p > 0$, then g is processed on the intervals obtained by using the bounds from the same inequalities.

Bounding Roots by Sturm Sequences

Consider a polynomial $f(t)$ defined on interval $[a, b]$. A *Sturm sequence* for f is a set of polynomials $f_i(t)$, $0 \leq i \leq m$, such that $\text{degree}(f_{i+1}) > \text{degree}(f_i)$ and the number of distinct real roots for f in $[a, b]$ is $N = s(a) - s(b)$, where $s(a)$ is the number of sign changes of $f_0(a), \dots, f_m(a)$ and $s(b)$ is the number of sign changes of $f_1(b), \dots, f_m(b)$. The total number of real-valued roots of f on $\mathbb{R}$ is $s(-\infty) - s(\infty)$. It is not always the case that $m = \text{degree}(f)$.

The classic Sturm sequence is $f_0(t) = f(t)$, $f_1(t) = f'(t)$, and $f_i(t) = -\text{remainder}(f_{i-2}/f_{i-1})$ for $i \geq 2$. The polynomials are generated by this method until the remainder term is a constant. An instructive example from the article by D. G. Hook and P. R. McAree in [Gla90] is $f(t) = t^3 + 3t^2 - 1$. The Sturm sequence is $f_0(t) = t^3 + 3t^2 - 1$, $f_1(t) = 3t^2 + 6t$, $f_2(t) = 2t + 1$, and $f_3 = 9/4$. Table 16.1 lists the signs of the Sturm polynomials for various t -values. Letting $N(a, b)$ denote the number of real-valued roots on the interval (a, b) , the table shows that

Table 16.1 Signs of the Sturm polynomials for $t^3 + 3t^2 - 1$ at various t -values.

t	$\text{Sign } f_0(t)$	$\text{Sign } f_1(t)$	$\text{Sign } f_2(t)$	$\text{Sign } f_3(t)$	Sign Changes
$-\infty$	—	+	—	+	3
-3	—	+	—	+	3
-2	+	0	—	+	2
-1	+	—	—	+	2
0	—	0	+	+	1
$+1$	+	+	+	+	0
$+\infty$	+	+	+	+	0

Table 16.2 Signs of the Sturm polynomials for $(t - 1)^3$ at various t -values.

t	$\text{Sign } f_0(t)$	$\text{Sign } f_1(t)$	$\text{Sign } f_2(t)$	Sign Changes
$-\infty$	—	+	0	1
0	—	+	0	1
$+\infty$	+	+	0	0

$N(-\infty, -3) = 0$, $N(-3, -2) = 1$, $N(-2, -1) = 0$, $N(-1, 0) = 1$, $N(0, 1) = 1$, and $N(1, \infty) = 0$. Moreover, the number of negative real roots is $N(-\infty, 0) = 2$, the number of positive real roots is $N(0, \infty) = 1$, and the total number of real roots is $N(-\infty, \infty) = 3$.

The next example shows that the number of polynomials in the Sturm sequence is not necessarily $\text{degree}(f) + 1$. The function $f(t) = (t - 1)^3$ has a Sturm sequence $f_0(t) = (t - 1)^3$, $f_1(t) = 3(t - 1)^2$, and $f_2(t) \equiv 0$ since f_1 exactly divides f_0 with no remainder. Table 16.2 lists sign changes for f at various t -values. The total number of real roots is $N(-\infty, \infty) = 1$.

16.5.2 METHODS IN MANY DIMENSIONS

Root finding in many dimensions is a more difficult problem than it is in one dimension. Two simple algorithms are summarized here: bisection and Newton's method.

Bisection

The bisection method for one dimension can be extended to multiple dimensions. Let $(f, g) : [a, b] \times [c, d] \rightarrow \mathbb{R}^2$. The problem is to find a point $(x, y) \in [a, b] \times [c, d]$ for which $(f(x, y), g(x, y)) = (0, 0)$. A quadtree decomposition of $[a, b] \times [c, d]$ can be used for the root search. Starting with the initial rectangle, f and g are evaluated at the four vertices:

- If either f or g has the same sign at the four vertices, the algorithm stops processing that region.
- If both f and g have a sign change at the vertices, they are evaluated at the center point of the region. If the values at the center are close enough to zero, that point is returned as a root and the search is terminated in that region.
- If the center value is not close enough to zero, the region is subdivided into four subregions by using the original four vertices, the midpoints of the four edges, and the center point. The algorithm is recursively applied to those four subregions.

It is possible that when a region is not processed further because f or g has the same sign at all four vertices, the region really does contain a root. The issue is the same as for one dimension—the initial rectangle needs to be partitioned to locate subrectangles on which a root is bound. The bisection method can be applied to each subrectangle that contains at least one root.

For three dimensions, an octree decomposition is applied in a similar way. For n dimensions, a 2^n -tree decomposition is used.

Newton's Method

Given differentiable $\mathbf{F} : \mathbb{R}^n \rightarrow \mathbb{R}^n$, the equation $\mathbf{F}(\mathbf{X}) = \mathbf{0}$ can be solved by the extension of Newton's method in one dimension. The iteration scheme that directly generalizes the method is to select an initial guess $(\mathbf{X}_0, \mathbf{F}(\mathbf{X}_0))$ and generate the next iterate by

$$\mathbf{X}_1 = \mathbf{X}_0 - (D\mathbf{F}(\mathbf{X}_0))^{-1}\mathbf{F}(\mathbf{X}_0)$$

The quantity $D\mathbf{F}(\mathbf{X})$ is the matrix of partial derivatives of $\mathbf{F}$, called the *Jacobian matrix*, and has entries $\partial F_i / \partial x_j$, where F_i is the i th component of $\mathbf{F}$ and x_j is the j th component of $\mathbf{X}$. Each iterate requires a matrix inversion. Although the obvious extension, it is not always the best to use. There are variations on the method that work much better in practice, some of which use *splitting methods* that avoid having to invert a matrix and usually have better convergence behavior.

16.6 INTEGRATION

Two standard methods are presented here for numerical integration, Romberg integration and Gaussian quadrature [BF01]. Either one is useful for graphics applications, for example, in computing the inverse arc length integral when reparameterizing by arc length.

16.6.1 ROMBERG INTEGRATION

Romberg integration is an excellent choice for numerical integration that is based on extrapolation methods and the trapezoid rule.

Richardson Extrapolation

The Richardson extrapolation method is very powerful. The key idea is to get high-order accuracy by using low-order formulas. Not only is it used in Romberg integration, but it is also used in the adaptive Runge-Kutta differential equation solvers.

Let Q be an unknown quantity approximated by $A(h)$ with approximation error of order $O(h^2)$. That is,

$$Q = A(h) + C_1 h^2 + O(h^4) = A(h) + O(h^2) \quad (16.1)$$

for some constant C_1 . This formula can be used to produce a (possibly) more accurate approximation. Replacing h by $h/2$ in the formula yields

$$Q = A\left(\frac{h}{2}\right) + \frac{C_1}{4} h^2 + O(h^4) \quad (16.2)$$

Taking 4 times Equation (16.2) and subtracting Equation (16.1), then dividing by 3 yields

$$Q = \frac{4A\left(\frac{h}{2}\right) - A(h)}{3} + O(h^4) \quad (16.3)$$

The goal is for the $O(h^4)$ terms in Equations (16.1) and (16.3) to be about the same size. If so, Equation (16.3) is more accurate since it does not have the h^2 term in it.

Define $A_1(h) = A(h)$ and $A_2(h) = (4A_1(h/2) - A_1(h))/3$. Other approximations can be written in an extrapolation table:

$$\begin{array}{cc}
A_1(h) & \\
A_1\left(\frac{h}{2}\right) & A_2(h) \\
A_1\left(\frac{h}{4}\right) & A_2\left(\frac{h}{2}\right) \\
A_1\left(\frac{h}{8}\right) & A_2\left(\frac{h}{4}\right) \\
\vdots & \vdots \\
A_1\left(\frac{h}{2^n}\right) & A_2\left(\frac{h}{2^{n-1}}\right)
\end{array}$$

The approximation $A_1(h/2^k)$ is of order $O(h^2)$, and the approximation $A_2(h/2^k)$ is of order $O(h^4)$.

If the original approximation is written as

$$Q = A(h) + C_1 h^2 + C_2 h^4 + O(h^6)$$

then the extrapolation table has an additional column:

$$\begin{array}{ccc}
A_1(h) & & \\
A_1\left(\frac{h}{2}\right) & A_2(h) & \\
A_1\left(\frac{h}{4}\right) & A_2\left(\frac{h}{2}\right) & A_3(h) \\
A_1\left(\frac{h}{8}\right) & A_2\left(\frac{h}{4}\right) & A_3\left(\frac{h}{2}\right) \\
\vdots & \vdots & \vdots \\
A_1\left(\frac{h}{2^n}\right) & A_2\left(\frac{h}{2^{n-1}}\right) & A_3\left(\frac{h}{2^{n-2}}\right)
\end{array}$$

where

$$A_3(h) = \frac{16A_2\left(\frac{h}{2}\right) - A_2(h)}{15}$$

The approximation $A_3(h/2^k)$ is of order $O(h^6)$.

In general, the extrapolation table is an $n \times m$ lower triangular matrix $T = [T_{rc}]$, where

$$T_{rc} = A_c \left(\frac{h}{2^{r-1}} \right)$$

and

$$A_c(h) = \frac{4^{c-1} A_{c-1} \left(\frac{h}{2} \right) - A_{c-1}(h)}{4^{c-1} - 1}$$

Trapezoid Rule

An approximation for $\int_a^b f(x) dx$ can be computed by first approximating $f(x)$ by the linear function

$$L(x) = \frac{x-b}{a-b} f(a) + \frac{x-a}{b-a} f(b)$$

and using $h[f(b) + f(a)]/2 = \int_a^b L(x) dx \doteq \int_a^b f(x) dx$. Some calculus shows that

$$\int_a^b f(x) dx = \frac{f(b) + f(a)}{2} h + O(h^3)$$

When $f(x) > 0$, the approximation is the area of a trapezoid with vertices at $(a, 0)$, $(a, f(a))$, $(b, 0)$, and $(b, f(b))$.

The integration interval $[a, b]$ can be divided into N subintervals over which the integration can be composited. Define $h = (b - a)/N$ and $x_j = a + jh$ for $0 \leq j \leq N$. It can be shown that

$$\int_a^b f(x) dx = \frac{h}{2} \left[f(a) + 2 \sum_{j=1}^{N-1} f(x_j) + f(b) \right] + O(h^2)$$

Note that the order of the approximation decreases by a power of one.

The Integration Method

Romberg integration uses the trapezoid rule to obtain preliminary approximations to the integral followed by Richardson extrapolation to obtain improvements.

Define $h_k = (b - a)/2^{k-1}$ for $k \geq 1$. The trapezoidal approximations corresponding to the interval partitions are

$$T_{k,1} = \frac{h_k}{2} \left[f(a) + 2 \left(\sum_{j=1}^{2^{k-1}-1} f(a + jh_k) \right) + f(b) \right]$$

and so

$$\int_a^b f(x) dx = T_{k,1} + O(h_k^2)$$

for all $k \geq 1$. The following recursion formula can be shown to hold:

$$2T_{k,1} = T_{k-1,1} + h_{k-1} \sum_{j=1}^{2^{k-2}} f(a + (j - 0.5)h_{k-1}) \quad (16.4)$$

for $k \geq 2$.

Richardson extrapolation can be applied; that is, generate the table

$$T_{i,j} = \frac{4^{j-1}T_{i,j-1} - T_{i-1,j-1}}{4^{j-1} - 1}$$

for $2 \leq j \leq i$. It can be shown that

$$\lim_{k \rightarrow \infty} T_{k,1} = \int_a^b f(x) dx \quad \text{if and only if} \quad \lim_{k \rightarrow \infty} T_{k,k} = \int_a^b f(x) dx$$

The second limit typically converges much faster than the first. The idea now is to choose a value n and use $T_{n,n}$ as an approximation to the integral. The code is

```
float RombergIntegral (float a, float b, float (*F)(float))
{
    const int order = 5;

    float rom[2][order];
    float h = b - a;

    // Initialize T_{1,1} entry.
    rom[0][0] = h * (F(a) + F(b))/2;

    for (int i = 2, ipower = 1; i <= order; i++, ipower *= 2, h /= 2)
    {
        // Calculate summation in recursion formula for T_{k,1}.
        float sum = 0;
        for (int j = 1; j <= ipower; j++)
            sum += F(a + h * (j - 0.5));
```

```

// trapezoidal approximations
rom[1][0] = (rom[0][0] + h * sum)/2;

// Richardson extrapolation
for (int k = 1, kpower = 4; k < i; k++, kpower *= 4)
    rom[1][k] = (kpower * rom[1][k - 1] - rom[0][k - 1])/(kpower - 1);

// Save extrapolated values for next pass.
for (j = 0; j < i; j++)
    rom[0][j] = rom[1][j];
}

return rom[0][order - 1];
}

```

The value `order` is arbitrarily chosen to be 5. Increasing the order will generally give better estimates, but at increased execution time. The values of $T_{i,j}$ are stored in `rom[2][order]`. Note that not all the values must be saved to build the next ones (so the first dimension of `rom` does not have to be `order`). This follows from the recursion given in Equation (16.4).

16.6.2 GAUSSIAN QUADRATURE

Gaussian quadrature approximates a definite integral,

$$\int_a^b f(x) dx \doteq \sum_{i=1}^n c_i f(x_i)$$

for some choice of constants c_i and values $x_i \in [a, b]$ regardless of what f is. If $a = -1$ and $b = 1$, the optimal choices for c_i and x_i are related to the Legendre polynomial of degree n . The x_i are the roots of that polynomial on $(-1, 1)$, and the c_i are given by

$$c_i = \int_{-1}^1 \prod_{j=1, j \neq i}^n \frac{x - x_j}{x_i - x_j} dx$$

The original problem can be transformed so that the quadrature formula applies. Let $t = (2x - a - b)/(b - a)$, then

$$\int_a^b f(x) dx = \int_{-1}^1 f\left(\frac{(b-a)t + b + a}{2}\right) \frac{b-a}{2} dt = \frac{b-a}{2} \int_{-1}^1 g(t) dt$$

The implementation of Gaussian quadrature amounts to selecting n , storing the tabulated values for c_i and x_i statically, and simply evaluating $((b-a)/2) \sum_{i=1}^n c_i g(t_i)$.

16.7 DIFFERENTIAL EQUATIONS

Differential equations are used to model physical systems that depend on rates of change of various quantities in the system. Ordinary differential equations have a single independent variable, usually time. Partial differential equations have multiple independent variables, usually including time and spatial variables. This section gives a brief overview of each type of equation.

16.7.1 ORDINARY DIFFERENTIAL EQUATIONS

A *first-order system of ordinary differential equations* is of the form

$$\frac{d\mathbf{X}(t)}{dt} = \mathbf{F}(t, \mathbf{X}(t))$$

where $\mathbf{F} : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$. The system is said to be *autonomous* if $\mathbf{F}$ does not depend explicitly on t , $\mathbf{F} = \mathbf{F}(\mathbf{X})$. An *initial value problem* supplies an initial condition $\mathbf{X}(t_0) = \mathbf{X}_0$ and time interval $t \geq t_0$. Under reasonable conditions on $\mathbf{F}$, the initial value problem has a unique solution for some range of t -values near the initial time t_0 . The system of equations is *explicit* in that the first-derivative term occurs explicitly in the equation (the left-hand side). An *implicit* equation is of the form $\mathbf{G}(t, \mathbf{X}, \mathbf{X}') = \mathbf{0}$.

A *second-order system of ordinary differential equations* is of the form

$$\frac{d^2\mathbf{X}(t)}{dt^2} = \mathbf{F}(t, \mathbf{X}(t), d\mathbf{X}(t)/dt)$$

An *initial value problem* supplies the initial conditions $\mathbf{X}(t_0) = \mathbf{X}_0$ and $d\mathbf{X}(t_0)/dt = \mathbf{D}_0$ and a time interval $t \geq t_0$. A *two-point boundary value problem* supplies an initial condition and final condition, $\mathbf{X}(t_0) = \mathbf{X}_0$ and $\mathbf{X}(t_1) = \mathbf{X}_1$ for $t \in [t_0, t_1]$. The system of equations is *explicit* in that the second-derivative term occurs explicitly in the equation. An *implicit* equation is of the form $\mathbf{G}(t, \mathbf{X}, \mathbf{X}', \mathbf{X}'') = \mathbf{0}$. It is possible to reduce a second-order system to a first-order system that has more variables. Setting $\mathbf{Y} = \mathbf{X}'$, the explicit system with n equations (the number of components of $\mathbf{X}$) is converted to a system with $2n$ equations, the number of components of the vector $\mathbf{Z} = (\mathbf{X}, \mathbf{Y})$:

$$\frac{d\mathbf{Z}}{dt} = \frac{d(\mathbf{X}, \mathbf{Y})}{dt} = \left(\frac{d\mathbf{X}}{dt}, \frac{d\mathbf{Y}}{dt} \right) = (\mathbf{Y}, \mathbf{F}(t, \mathbf{X}, \mathbf{Y})) = \mathbf{G}(t, \mathbf{Z})$$

The solution $\mathbf{Z}(t)$ has $2n$ components, but only the first n matter for the original problem. The last n just list the derivatives of the first n .

Second-order equations arise in physics problems, a topic that has gained a lot of popularity in games. Realistic collision response requires solving second-order, initial value differential equations, so it is necessary to understand what these equations are and how to solve them. The rest of this section presents a few standard methods for solving initial value systems. A problem such as minimum weight pathfinding—for example, the shortest distance between two points on a surface—requires solving second-order, boundary value differential equations. This is a more difficult problem to solve and requires either *shooting methods* or *relaxation methods*, which are not discussed here (see [BF01]).

Euler's Method

The easiest differential equation solver is Euler's method. The initial value is $(t_0, \mathbf{X}_0)$. Successive approximations $(t_i, \mathbf{X}_i)$ for $i > 0$ are generated by using a first-order forward difference to approximate the first derivative:

$$\mathbf{X}_{i+1} = \mathbf{X}_i + h\mathbf{F}(t_i, \mathbf{X}_i)$$

$$t_{i+1} = t_i + h$$

where $h > 0$ is a sufficiently small step size.

Midpoint Method

The midpoint method is a second-order Runge-Kutta algorithm. The initial value is $(t_0, \mathbf{X}_0)$. Successive approximations $(t_i, \mathbf{X}_i)$ for $i > 0$ are generated by

$$\mathbf{A}_1 = \mathbf{F}(t_i, \mathbf{X}_i)$$

$$\mathbf{A}_2 = \mathbf{F}(t_i + h/2, \mathbf{X}_i + h\mathbf{A}_1/2)$$

$$\mathbf{X}_{i+1} = \mathbf{X}_i + h\mathbf{A}_2$$

$$t_{i+1} = t_i + h$$

Runge-Kutta Fourth-Order Method

The Runge-Kutta fourth-order method has good accuracy. The initial value is $(t_0, \mathbf{X}_0)$. Successive approximations $(t_i, \mathbf{X}_i)$ for $i > 0$ are generated by

$$\mathbf{A}_1 = \mathbf{F}(t_i, \mathbf{X}_i)$$

$$\mathbf{A}_2 = \mathbf{F}(t_i + h/2, \mathbf{X}_i + h\mathbf{A}_1/2)$$

$$\mathbf{A}_3 = \mathbf{F}(t_i + h/2, \mathbf{X}_i + h\mathbf{A}_2/2)$$

$$\mathbf{A}_4 = \mathbf{F}(t_i + h, \mathbf{X}_i + h\mathbf{A}_3)$$

$$\mathbf{X}_{i+1} = \mathbf{X}_i + \frac{h}{6}(\mathbf{A}_1 + 2\mathbf{A}_2 + 2\mathbf{A}_3 + \mathbf{A}_4)$$

$$t_{i+1} = t_i + h$$

Runge-Kutta with Adaptive Step

For some data sets, it is possible to dynamically adjust the step size h to reduce the total number of steps to get to a desired final time. The following algorithm is fifth order and adjusts the step size accordingly.

1. Take two half-steps:

$$\mathbf{A}_1 = \mathbf{F}(t_i, \mathbf{X}_i)$$

$$\mathbf{A}_2 = \mathbf{F}(t_i + h/4, \mathbf{X}_i + h\mathbf{A}_1/4)$$

$$\mathbf{A}_3 = \mathbf{F}(t_i + h/4, \mathbf{X}_i + h\mathbf{A}_2/4)$$

$$\mathbf{A}_4 = \mathbf{F}(t_i + h/2, \mathbf{X}_i + h\mathbf{A}_3/2)$$

$$\mathbf{X}_{\text{inter}} = \mathbf{X}_i + \frac{h}{12}(\mathbf{A}_1 + 2\mathbf{A}_2 + 2\mathbf{A}_3 + \mathbf{A}_4)$$

and

$$\mathbf{B}_1 = \mathbf{F}(t_i + h/2, \mathbf{X}_{\text{inter}})$$

$$\mathbf{B}_2 = \mathbf{F}(t_i + 3h/4, \mathbf{X}_{\text{inter}} + h\mathbf{B}_1/4)$$

$$\mathbf{B}_3 = \mathbf{F}(t_i + 3h/4, \mathbf{X}_{\text{inter}} + h\mathbf{B}_2/4)$$

$$\mathbf{B}_4 = \mathbf{F}(t_i + h, \mathbf{X}_{\text{inter}} + h\mathbf{B}_3/2)$$

$$\mathbf{X}_{\text{half}} = \mathbf{X}_{\text{inter}} + \frac{h}{12}(\mathbf{B}_1 + 2\mathbf{B}_2 + 2\mathbf{B}_3 + \mathbf{B}_4)$$

2. Take a full step:

$$\mathbf{C}_1 = \mathbf{F}(t_i, \mathbf{X}_n)$$

$$\mathbf{C}_2 = \mathbf{F}(t_i + h/2, \mathbf{X}_i + h\mathbf{C}_1/2)$$

$$\mathbf{C}_3 = \mathbf{F}(t_i + h/2, \mathbf{X}_i + h\mathbf{C}_2/2)$$

$$\mathbf{C}_4 = \mathbf{F}(t_i + h, \mathbf{X}_i + h\mathbf{C}_3)$$

$$\mathbf{X}_{\text{full}} = \mathbf{X}_i + \frac{h}{6}(\mathbf{C}_1 + 2\mathbf{C}_2 + 2\mathbf{C}_3 + \mathbf{C}_4)$$

3. Compute the fractional error term Δ , where $\varepsilon > 0$ is a constant specified by the user:

$$\Delta = \frac{1}{\varepsilon} \max_i \left| \frac{(\mathbf{X}_{\text{half}})_i - (\mathbf{X}_{\text{full}})_i}{hF_i(t_n, x_n) + \epsilon_0} \right|$$

where ϵ_0 is a very small positive number that protects against the case $F_i(t_n, x_n) = 0$, in which case Δ becomes a very large positive number. The choice of ε is crucial. Its value can be selected by experimentation in a specific application.

4. If $\Delta \leq 1$, then the iteration is successful. The step size h is used and the next iterates are

$$\mathbf{X}_{i+1} = \mathbf{X}_{\text{half}} + \frac{1}{15}(\mathbf{X}_{\text{half}} - \mathbf{X}_{\text{full}})$$

$$t_{i+1} = t_i + h$$

The successful iteration suggests trying a larger step size for the next iteration. The step size is adjusted as follows. Let $S < 1$ be a number close to 1 (typical $S = 0.9$). If $\Delta > (S/4)^5$, then a conservative increase is made: $h \leftarrow Sh\Delta^{-1/5}$. If $\Delta \leq (S/4)^5$, a more aggressive increase is made: $h \leftarrow 4h$.

5. If $\Delta > 1$, then the iteration fails. The step size must be reduced and the iteration is repeated starting with the initial iterate x_n . The adjustment is $h \leftarrow Sh\Delta^{-1/4}$. Repeat step 1 with this new step size. A check must be made for the low-probability case where $h \rightarrow 0$.

16.7.2 PARTIAL DIFFERENTIAL EQUATIONS

Ordinary differential equations involve specifying changes to a function of one independent variable, $\mathbf{X}(t)$. Partial differential equations are a natural extension to handle functions of many independent variables. Although this topic is immense, it is relevant to games because second-order, linear partial differential equations arise

naturally in modeling physical phenomena. On a hardware platform with a lot of processing power, it is possible to procedurally morph geometric data based on the relevant physics. For example, the flapping of a flag in the wind can be modeled as a wavelike behavior. A partial differential equation can be used to model the motion, and a numerical solution can be computed at run time.

The second-order partial differential equations are characterized as parabolic, hyperbolic, or elliptic. Let $x \in \mathbb{R}$, $t \geq 0$, and $u = u(x, t) \in \mathbb{R}$ in the following examples.

Parabolic: Heat Transfer, Population Dynamics

Diffusion of heat $u(x, t)$ in a rod of length L and with heat source $f(x)$ is modeled by

$$\begin{aligned} u_t(x, t) &= u_{xx}(x, t) + f(x), & x \in (0, L), t > 0 & \quad (\text{from conservation laws}) \\ u(x, 0) &= g(x), & x \in [0, L] & \quad (\text{initial heat distribution}) \\ u(0, t) &= a(t), u(L, t) = b(t), t \geq 0 & & \quad (\text{temperature at boundaries}) \end{aligned}$$

or

$$u_x(0, t) = u_x(L, t) = 0, \quad t \geq 0 \quad (\text{insulated boundaries})$$

Numerical solution for the case of no heat source, $f = 0$, and insulated boundaries uses finite differences to approximate the partial derivatives. Select $m + 1$ spatial locations uniformly sampled as $x_i = i \Delta x$ for $0 \leq i \leq m$ with $\Delta x = L/m$. Select temporal samples as $t_j = j \Delta t$ for $j \geq 0$ with $\Delta t > 0$. The estimates of temperature are $u_i^{(j)} \doteq u(x_i, t_j)$ for $0 \leq i \leq m$ and $j \geq 0$. The sampled initial temperature is $g_i = g(x_i)$ for $0 \leq i \leq m$. Approximate the time derivative by a forward difference:

$$u_t(x, t) \doteq \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t}$$

and approximate the spatial derivatives by central differences:

$$u_{xx}(x, t) \doteq \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{(\Delta x)^2}$$

Replace these in the heat equation to obtain

$$\frac{u_i^{(j+1)} - u_i^{(j)}}{\Delta t} = \frac{u_{i+1}^{(j)} - 2u_i^{(j)} + u_{i-1}^{(j)}}{(\Delta x)^2}$$

The boundary conditions are $u_0^{(j)} = u_m^{(j)} = 0$ for $j \geq 0$. The numerical algorithm is implemented as

$$\begin{aligned}
u_i^{(0)} &= g_i, & 0 \leq i \leq m \\
u_0^{(j)} &= u_m^{(j)} = 0, & j \geq 0 \\
u_i^{(j+1)} &= u_i^{(j)} + \frac{\Delta t}{(\Delta x)^2} \left(u_{i+1}^{(j)} - 2u_i^{(j)} + u_{i-1}^{(j)} \right), & 1 \leq i \leq m-1, \quad j \geq 0
\end{aligned}$$

For this to be numerically stable, $\Delta t < (\Delta x)^2/2$ is required. An alternative scheme is the Crank-Nicholson method:

$$u_i^{(j+1)} = u_i^{(j)} + \frac{\Delta t}{(\Delta x)^2} \left(\frac{u_{i+1}^{(j)} - 2u_i^{(j)} + u_{i-1}^{(j)}}{2} + \frac{u_{i+1}^{(j+1)} - 2u_i^{(j+1)} + u_{i-1}^{(j+1)}}{2} \right)$$

This method is stable for all $\Delta t > 0$ but is harder to solve since $u_i^{(j+1)}$ is implicitly defined.

Hyperbolic: Wave and Shock Phenomena

Displacement $u(x, t)$ of an elastic string is modeled by

$$\begin{aligned}
u_{tt}(x, t) &= u_{xx}(x, t), \quad x \in (0, L), \quad t > 0 \quad (\text{from conservation laws}) \\
u(x, 0) &= f(x), \quad u_t(x, 0) = g(x), \quad x \in [0, L] \quad (\text{initial displacement and speed}) \\
u(0, t) &= a(t), \quad u(L, t) = b(t), \quad t \geq 0 \quad (\text{location of string ends})
\end{aligned}$$

Numerical solution for the case of clamped ends, $a = 0$ and $b = 0$, uses finite differences to approximate the partial derivatives. Centralized differences are used for both the time and spatial derivatives:

$$\begin{aligned}
u_i^{(0)} &= f_i, & 0 \leq i \leq m \\
u_i^{(1)} &= u_i^{(0)} + (\Delta t)g_i, & 0 \leq i \leq m \\
u_0^{(j)} &= u_m^{(j)} = 0, & j \geq 0 \\
\frac{u_i^{(j+1)} - 2u_i^{(j)} + u_i^{(j-1)})}{(\Delta t)^2} &= \frac{u_{i+1}^{(j)} - 2u_i^{(j)} + u_{i-1}^{(j)})}{(\Delta x)^2}, & 1 \leq i \leq m-1, \quad j \geq 1
\end{aligned}$$

The method is stable when $\Delta t < \Delta h$. If the right-hand side is modified as in the Crank-Nicholson method for the heat equation, then the method is stable for all $\Delta t > 0$.

Elliptic: Steady-State Heat Flow, Potential Theory

Steady-state distribution of heat $u(x)$ in a bar of length L with heat source $f(x)$ is modeled by

$$u_{xx}(x) = -f(x), \quad x \in (0, L) \quad (t \rightarrow \infty \text{ in the heat equation})$$

$$u(0) = A, u(L) = B, \quad (\text{boundary conditions})$$

The numerical method for the constant temperature boundary, $A = B = 0$, is

$$u_0 = 0, u_m = 0$$

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2} = -f_i, \quad 1 \leq i \leq m-1$$

Define the $(m-1) \times 1$ vectors $\mathbf{u} = [u_i]$, where $1 \leq i \leq m-1$ and $\mathbf{b} = [-(\Delta x)^2 f_i]$. This vector is the unknown in a linear system $A\mathbf{u} = \mathbf{b}$, where A is tridiagonal with main diagonal -2 and sub- and superdiagonals 1 . Such systems are solved robustly in $O(m)$ time.

Extension to Higher Dimensions

Consider $u(x, y, t)$ for 2D problems. The heat equation is $u_t = u_{xx} + u_{yy}$; the wave equation is $u_{tt} = u_{xx} + u_{yy}$; and the potential equation is $u_{xx} + u_{yy} = f(x, y)$. If the domain for (x, y) is a rectangle, then finite difference methods such as the ones used in the 1D problems extend fairly easily. If the domain is not rectangular, then *finite elements* must be used—approximating the boundary of the domain by a polygon, then decomposing the polygon into triangles. A good method for the decomposition is by Narkhede and Manocha [Pae95].

For example, consider $u_{xx} + u_{yy} = 0$, where domain R is not rectangular. Let $u(x, y)$ be specified on the boundary of R . Decompose region R into triangles. On each triangle approximate the true solution $u(x, y)$ by a linear function $v(x, y)$ that interpolates the triangle vertices. If the vertices are $\mathbf{P}_i = (x_i, y_i, v_i)$ for $0 \leq i \leq 2$ and where the v_i estimate $u(x_i, y_i)$, then a triangle normal is $\mathbf{N} = (\mathbf{P}_1 - \mathbf{P}_0) \times (\mathbf{P}_2 - \mathbf{P}_0)$, and $v(x, y)$ is the linear function determined by $\mathbf{N} \cdot ((x, y, v(x, y)) - \mathbf{V}_0) = 0$. The boundary v_i are known, but the interior v_i must be determined.

Solving the potential equation on R is equivalent to finding a function u that minimizes the integral

$$I = \int \int_R u_x^2 + u_y^2 dx dy$$

subject to the boundary conditions. Define $\tilde{I}$ to be the approximate integral where $u(x, y)$ is replaced by $v(x, y)$. For triangle T , let the linear approximation for u

on that triangle be denoted $v_T(x, y) = \alpha_T x + \beta_T y + \gamma_T$; then the approximating integral to I is

$$\tilde{I} = \sum_T (\alpha_T^2 + \beta_T^2) \text{Area}(T)$$

Since α_T and β_T are linear functions of the interior values v_i , $\tilde{I}$ is quadratic in v_i . Minimizing a quadratic function can be done by solving a linear system (set derivatives equal to zero) or by the conjugate gradient method (equivalent to solving the linear system, but uses root-finding techniques).

16.8 FAST FUNCTION EVALUATION

A handful of functions that are typically expensive to compute occur frequently in computer graphics and games applications: computing the length of a vector (requires a square root); resizing a vector to be unit length (requires a reciprocal square root); computing angles from spatial information (requires inverse tangent or another inverse trigonometric function); and computing sine or cosine. Even division by a floating-point number is somewhat expensive compared to additions and multiplications. Current-generation CPUs have added fast hardware support for many of these operations, most notably inverse square root and fast (but less accurate) division.

The algorithms described in this section are designed for fast evaluation of various functions. Some of these algorithms can be implemented in hardware, but they can be implemented easily in software and might provide an alternative to the operations provided by a floating-point coprocessor whose function calls still take a significant number of cycles.

A wonderful source for tricks and techniques for mathematical functions is the [AS65]. In particular, there are lots of formulas for approximating functions by polynomials of small degree. The formulas are always accompanied by a domain on which the approximation is intended and a global error estimate for that domain.

16.8.1 SQUARE ROOT AND INVERSE SQUARE ROOT

Many of the fast square root methods provide a low-accuracy result, but for many graphics applications, this is an acceptable trade-off. [Gla90] has a number of articles on these methods.

A method by Paul Lalonde and Robert Dawson represents a nonnegative, floating-point number as $x = n \cdot 2^{2p}$, where p is an integer and $m \in [1, 4)$ is the mantissa. Thus, $\sqrt{x} = \sqrt{m \cdot 2^{2p}} = \sqrt{m} \cdot 2^p$, where $\sqrt{m} \in [1, 2)$. Using an n -bit mantissa, a table of values for $\sqrt{m}$ can be computed and stored for lookup. The pseudo-code is


```

float SquareRoot (float x)
{
    SplitFloat(x,p,m); // p = power, m = mantissa
    p = p/2;
    m = Lookup[m];
    return MakeFloat(p,m);
}

```

Steve Hill in [Arv91] provides code to implement this using IEEE double-precision, floating-point numbers. At the expense of one division, this routine can be used to compute the inverse square root of y by $1/\sqrt{y} = \sqrt{1/y} = \sqrt{x}$, where $x = 1/y$.

[Pae95] has an algorithm by Ken Turkowski that uses Newton's method for computing the inverse square root. Only a few iterations are used, and an initial point is provided by table lookup just as in the method for square root calculation. If $y = 1/\sqrt{x}$, then $1/y^2 - x = 0$. Define $f(y) = 1/y^2 - x$ for the selected x . A positive root $\bar{y}$ to $f(y) = 0$ will be the inverse square root of x . The equation can be solved by Newton iteration. An initial guess $y_0 > 0$ is selected. The iterates are generated by

$$y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)} = \frac{y_i(3 - xy_i^2)}{2}$$

The initial guess is chosen just as for the square root algorithm mentioned earlier. The mantissa is used to index into a table of inverse square root quantities. The value looked up is polished further by the iteration just mentioned. The smaller the lookup table, the larger the number of iterations to get to the desired accuracy. Once the inverse square root $r = 1/\sqrt{x}$ is computed, the square root may be obtained by an extra multiplication, $\sqrt{x} = xr$.

Bit hacks may be used by interpreting a 32-bit floating-point number as a 32-bit integer, or a 64-bit floating-point number as a 64-bit integer, and then applying a single Newton iteration as described earlier. The origin of this bit hack (for 32-bit numbers) is attributed to Gary Tivoli, but it was unclear how a certain magic number was selected. A detailed description of the selection (for both 32-bit and 64-bit numbers) is provided in the online document [Lom03]. Implementations for 32-bit and 64-bit numbers is included in the source code on the CD-ROM.

16.8.2 SINE, COSINE, AND TANGENT

Because sine and cosine are bounded functions with not much variation, the simplest method for fast evaluation of sine and cosine is to use range reduction followed by a table lookup. Both tables store values in the range $[0, \pi/2]$.

Polynomial approximations can also be used [AS65]. Approximations to sine on the interval $[0, \pi/2]$ are

$$\sin(x) = \sum_{i=0}^2 a_i x^{2i+1} + \epsilon(x)$$

where $a_0 = 1$, $a_1 = -1.6605e - 01$, $a_2 = 7.61e - 03$, and $|\epsilon(x)| \leq 1.6415e - 04$; and

$$\sin(x) = \sum_{i=0}^5 a_i x^{2i+1} + \epsilon(x)$$

where $a_0 = 1$, $a_1 = -1.666666664e - 01$, $a_2 = 8.3333315e - 03$, $a_3 = -1.984090e - 04$, $a_4 = 2.7526e - 06$, $a_5 = -2.39e - 08$, and $|\epsilon(x)| \leq 2.3279e - 09$.

Approximations to cosine on the interval $[0, \pi/2]$ are

$$\cos(x) = \sum_{i=0}^2 a_i x^{2i} + \epsilon(x)$$

where $a_0 = 1$, $a_1 = -4.9670e - 01$, $a_2 = 3.705e - 02$, and $|\epsilon(x)| \leq 1.188e - 03$; and

$$\cos(x) = \sum_{i=0}^5 a_i x^{2i} + \epsilon(x)$$

where $a_0 = 1$, $a_1 = -4.999999963e - 01$, $a_2 = 4.16666418e - 02$, $a_3 = -1.3888397e - 03$, $a_4 = 2.47609e - 05$, $a_5 = -2.605e - 07$, and $|\epsilon(x)| \leq 2.3082e - 09$.

Approximations to tangent on the interval $[0, \pi/4]$ are

$$\tan(x) = \sum_{i=0}^2 a_i x^{2i+1} + \epsilon(x)$$

where $a_0 = 1$, $a_1 = 3.1755e - 01$, $a_2 = 2.0330e - 01$, and $|\epsilon(x)| \leq 8.0613e - 04$; and

$$\tan(x) = \sum_{i=0}^6 a_i x^{2i+1} + \epsilon(x)$$

where $a_0 = 1$, $a_1 = 3.333314036e - 01$, $a_2 = 1.333923995e - 01$, $a_3 = 5.33740603e - 02$, $a_4 = 2.45650893e - 02$, $a_5 = 2.9005250e - 03$, $a_6 = 9.5168091e - 03$, and $|\epsilon(x)| \leq 1.8897e - 08$.

16.8.3 INVERSE TANGENT

A family of polynomials that approximate the inverse tangent function can be built using a least-squares algorithm based on integrals (as compared to the summations that occur in Section 16.3). The approximations are computed to $\tan^{-1}(z)$ for $z \in [-1, 1]$. For $z > 1$, the trigonometric identity $\tan^{-1}(z) = \pi/2 - \tan^{-1}(1/z)$ reduces the problem to evaluating the inverse tangent for $u = 1/z \in (-1, 1)$. Similarly, for $z < -1$, $\tan^{-1}(z) = -\pi/2 - \tan^{-1}(1/z)$.

The function $\tan^{-1}(z)$ is an odd function on $[-1, 1]$. An approximating polynomial $p(z) = \sum_{i=0}^n a_i z^{2i+1}$ is desired; this is also an odd function. Ideally, n can be

chosen to be a small number so that only a few polynomial terms have to be computed using additions and multiplications. The size of n , of course, will depend on how much error an application can tolerate. The idea is to select the coefficients $\mathbf{a} = [a_i]$ to minimize the squared integral error:

$$E(\mathbf{a}) = \int_{-1}^1 [p(z; \mathbf{a}) - \tan^{-1}(z)]^2 dz$$

The minimum must occur when $\nabla E = \mathbf{0}$. The i th derivative is

$$\begin{aligned} \frac{\partial E}{\partial a_i} &= \int_{-1}^1 2z^{2i+1} [p(z; \mathbf{a}) - \tan^{-1}(z)] dz \\ &= 2 \int_{-1}^1 z^{2i+1} p(z; \mathbf{a}) dz - 2 \int_{-1}^1 z^{2i+1} \tan^{-1}(z) dz \\ &= 4 \sum_{j=0}^n \frac{a_j}{2i+2j+3} - \frac{2}{i+1} \left(\frac{\pi}{4} (1 + (-1)^i) - \sum_{j=0}^i \frac{(-1)^j}{2i+1-2j} \right) \end{aligned}$$

The $n+1$ equations obtained from $\nabla E = \mathbf{0}$ provide a linear system in the $n+1$ unknowns, the components of $\mathbf{a}$. This system can be solved by standard solvers. It is also possible to obtain global error bounds from the theory of Taylor series (estimating error when a series is truncated). The coefficients and global error bounds are summarized for $2 \leq n \leq 5$ in Table 16.3.

Table 16.3 Coefficients for polynomial approximations to $\tan^{-1}(z)$.

n	Coefficients			Maximum Error
2	$a_0 = +0.995987$	$a_1 = -0.292298$	$a_2 = +0.0830353$	$1.32603e-03$
3	$a_0 = +0.999337$ $a_3 = -0.0410731$	$a_1 = -0.322456$	$a_2 = +0.149384$	$2.05811e-04$
4	$a_0 = +1.000000$ $a_3 = -0.0956074$	$a_1 = -0.332244$ $a_4 = +0.0257527$	$a_2 = +0.187557$	$6.53509e-05$
5	$a_0 = +1.000700$ $a_3 = -0.317300$	$a_1 = -0.347418$ $a_4 = +0.259954$	$a_2 = +0.278742$ $a_5 = -0.0894795$	$1.95831e-04$



ROTATIONS

Rotation of vectors is a common operation in computer graphics. Most programmers are comfortable with rotation matrices. A few are uncomfortable with the quaternions, an alternate representation for rotation matrices, because of their inherent mathematical nature. This chapter summarizes both topics, but in the end the emphasis is on the comparison of rotation matrices and quaternions regarding memory usage and computational speed.

17.1 ROTATION MATRICES

A 2D rotation of the vector (x, y) to the vector (x', y') by an angle θ is represented by the matrix

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = R_2 \begin{bmatrix} x \\ y \end{bmatrix} \quad (17.1)$$

where the last equality defines the rotation matrix R_2 . The rotation direction is counterclockwise in the xy -plane when $\theta > 0$. We saw such a rotation in Figure 2.3.

Using the rotation matrix in Equation (17.1) as motivation, a 3D rotation of the vector (x, y, z) to the vector (x', y', z') by an angle θ about the z -axis is represented by the matrix

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = R_3 \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (17.2)$$

where the last equality defines the rotation matrix R_3 . The idea is to rotate the 2D component of the vectors in the xy -plane using the rotation matrix of Equation (17.1).

17.1.1 AXIS/ANGLE TO MATRIX

We may use the matrix of Equation (17.2) to create a matrix representing a general rotation. Let the rotation axis have direction $\mathbf{W}$, a unit-length vector. Let θ be the angle of rotation. Choose any pair of unit-length vectors $\mathbf{U}$ and $\mathbf{V}$ so that the set $\{\mathbf{U}, \mathbf{V}, \mathbf{W}\}$ is a right-handed orthonormal set. That is, the vectors are mutually perpendicular and $\mathbf{W} = \mathbf{U} \times \mathbf{V}$. A vector $\mathbf{P}$ will be rotated to a vector $\mathbf{Q}$. We may represent $\mathbf{P}$ as

$$\mathbf{P} = x\mathbf{U} + y\mathbf{V} + z\mathbf{W}$$

where $x = \mathbf{U} \cdot \mathbf{P}$, $y = \mathbf{V} \cdot \mathbf{P}$, and $z = \mathbf{W} \cdot \mathbf{P}$. Similarly, we may represent $\mathbf{Q}$ as

$$\mathbf{Q} = x'\mathbf{U} + y'\mathbf{V} + z'\mathbf{W}$$

where $x' = \mathbf{U} \cdot \mathbf{Q}$, $y' = \mathbf{V} \cdot \mathbf{Q}$, and $z' = \mathbf{W} \cdot \mathbf{Q}$. The matrix of Equation (17.2) is applied to (x, y, z) to obtain (x', y', z') . Since only the x - and y -components change, the rotation is necessarily about the $\mathbf{W}$ -axis.

Here are some algebraic computations to start the process of constructing the rotation matrix R for which $\mathbf{Q} = R\mathbf{P}$. Starting with Equation (17.2), replace x , y , z , x' , y' , and z' with the dot products mentioned previously:

$$\begin{bmatrix} \mathbf{U} \cdot \mathbf{Q} \\ \mathbf{V} \cdot \mathbf{Q} \\ \mathbf{W} \cdot \mathbf{Q} \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{U} \cdot \mathbf{P} \\ \mathbf{V} \cdot \mathbf{P} \\ \mathbf{W} \cdot \mathbf{P} \end{bmatrix}$$

This equation factors to

$$\begin{bmatrix} \mathbf{U}^T \\ \mathbf{V}^T \\ \mathbf{W}^T \end{bmatrix} \mathbf{Q} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{U}^T \\ \mathbf{V}^T \\ \mathbf{W}^T \end{bmatrix} \mathbf{P}$$

The matrix involving the transposes of $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ is itself a rotation matrix. Applying its transpose to both sides of the equation, we have

$$\mathbf{Q} = [\mathbf{U} \quad \mathbf{V} \quad \mathbf{W}] \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{U}^T \\ \mathbf{V}^T \\ \mathbf{W}^T \end{bmatrix} \mathbf{P}$$

The rotation matrix R is the product of the three matrices in front of $\mathbf{P}$:

$$\begin{aligned}
R &= [\mathbf{U} \quad \mathbf{V} \quad \mathbf{W}] \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{U}^T \\ \mathbf{V}^T \\ \mathbf{W}^T \end{bmatrix} \\
&= [\mathbf{U} \quad \mathbf{V} \quad \mathbf{W}] \begin{bmatrix} (\cos \theta)\mathbf{U}^T - (\sin \theta)\mathbf{V}^T \\ (\sin \theta)\mathbf{U}^T + (\cos \theta)\mathbf{V}^T \\ \mathbf{W}^T \end{bmatrix} \\
&= \mathbf{U} \left((\cos \theta)\mathbf{U}^T - (\sin \theta)\mathbf{V}^T \right) + \mathbf{V} \left((\sin \theta)\mathbf{U}^T + (\cos \theta)\mathbf{V}^T \right) + \mathbf{W}\mathbf{W}^T \\
&= (\cos \theta) \left(\mathbf{U}\mathbf{U}^T + \mathbf{V}\mathbf{V}^T \right) + (\sin \theta) \left(\mathbf{V}\mathbf{U}^T - \mathbf{U}\mathbf{V}^T \right) + \mathbf{W}\mathbf{W}^T
\end{aligned} \tag{17.3}$$

The rotation matrix should depend only on the rotation axis direction $\mathbf{W}$ and the rotation angle θ . Eliminating the dependency on choice of $\mathbf{U}$ and $\mathbf{V}$ requires the following identities. First,

$$\begin{aligned}
\mathbf{P} &= x\mathbf{U} + y\mathbf{V} + z\mathbf{W} \\
&= (\mathbf{U} \cdot \mathbf{P})\mathbf{U} + (\mathbf{V} \cdot \mathbf{P})\mathbf{V} + (\mathbf{W} \cdot \mathbf{P})\mathbf{W} \\
&= \mathbf{U}\mathbf{U}^T\mathbf{P} + \mathbf{V}\mathbf{V}^T\mathbf{P} + \mathbf{W}\mathbf{W}^T\mathbf{P} \\
&= \left(\mathbf{U}\mathbf{U}^T + \mathbf{V}\mathbf{V}^T + \mathbf{W}\mathbf{W}^T \right) \mathbf{P}
\end{aligned}$$

This is true no matter what choice for $\mathbf{P}$, which implies the matrix identity

$$\mathbf{U}\mathbf{U}^T + \mathbf{V}\mathbf{V}^T + \mathbf{W}\mathbf{W}^T = \mathbf{I} \tag{17.4}$$

where $\mathbf{I}$ is the identity matrix. Second, let $\mathbf{W} = (w_0, w_1, w_2)$ and define the skew-symmetric matrix

$$S = \begin{bmatrix} 0 & -w_2 & w_1 \\ w_2 & 0 & -w_0 \\ -w_1 & w_0 & 0 \end{bmatrix} \tag{17.5}$$

then

$$\begin{aligned}
S\mathbf{P} &= \mathbf{W} \times \mathbf{P} \\
&= \mathbf{W} \times (x\mathbf{U} + y\mathbf{V} + z\mathbf{W}) \\
&= x\mathbf{V} - y\mathbf{U} \\
&= (\mathbf{U} \cdot \mathbf{P})\mathbf{V} - (\mathbf{V} \cdot \mathbf{P})\mathbf{U} \\
&= \left(\mathbf{V}\mathbf{U}^T - \mathbf{U}\mathbf{V}^T \right) \mathbf{P}
\end{aligned}$$

This is true no matter what choice for $\mathbf{P}$, which implies the matrix identity

$$\mathbf{V}\mathbf{U}^T - \mathbf{U}\mathbf{V}^T = S \quad (17.6)$$

Third,

$$\begin{aligned} S^2\mathbf{P} &= \mathbf{W} \times (\mathbf{W} \times \mathbf{P}) \\ &= \mathbf{W} \times (x\mathbf{V} - y\mathbf{U}) \\ &= -(x\mathbf{U} + y\mathbf{V}) \\ &= -((\mathbf{U} \cdot \mathbf{P})\mathbf{U} + (\mathbf{V} \cdot \mathbf{P})\mathbf{V}) \\ &= -(\mathbf{U}\mathbf{U}^T + \mathbf{V}\mathbf{V}^T)\mathbf{P} \end{aligned}$$

This is true no matter what choice for $\mathbf{P}$, which implies the matrix identity

$$\mathbf{U}\mathbf{U}^T + \mathbf{V}\mathbf{V}^T = -S^2 \quad (17.7)$$

Replacing Equations (17.4), (17.6), and (17.7) into Equation (17.3) produces

$$\begin{aligned} R &= (\cos \theta) (\mathbf{U}\mathbf{U}^T + \mathbf{V}\mathbf{V}^T) + (\sin \theta) (\mathbf{V}\mathbf{U}^T - \mathbf{U}\mathbf{V}^T) + \mathbf{W}\mathbf{W}^T \\ &= (\cos \theta)(-S^2) + (\sin \theta)(S) + (I + S^2) \\ &= I + (\sin \theta)S + (1 - \cos \theta)S^2 \end{aligned} \quad (17.8)$$

When applied to the vector $\mathbf{P}$,

$$R\mathbf{P} = \mathbf{P} + (\sin \theta)\mathbf{W} \times \mathbf{P} + (1 - \cos \theta)\mathbf{W} \times (\mathbf{W} \times \mathbf{P}) \quad (17.9)$$

17.1.2 MATRIX TO AXIS/ANGLE

Given a rotation matrix R , sometimes you might want to know the axis of rotation and a rotation angle. The answer is not unique. If an axis direction is $\mathbf{W}$ and the rotation angle is θ , then another axis direction is $-\mathbf{W}$ and the corresponding rotation angle is $-\theta$. Also, $\theta + 2\pi k$ is a rotation angle for any integer k . Despite this, it is always possible to extract some axis direction and some rotation angle.

The *trace* of a matrix is defined to be the sum of its diagonal terms. Some algebra applied to the matrix in Equation (17.8) will show that $\cos \theta = (\text{Trace}(R) - 1)/2$, in which case

$$\theta = \cos^{-1} ((\text{Trace}(R) - 1)/2) \in [0, \pi]$$

Also, it is easily shown that

$$R - R^T = (2 \sin \theta)S$$

If $\theta = 0$, then $R = R^T$ and any unit-length direction vector for the axis is valid since there is no rotation. If $\theta \in (0, \pi)$, an axis may be extracted from the equation $R - R^T = (2 \sin \theta)S$. Let $S = [s_{ij}]$, where $s_{00} = s_{11} = s_{22} = 0$, $s_{10} = -s_{01}$, $s_{20} = -s_{02}$, and $s_{21} = -s_{12}$; then

$$\mathbf{W} = \frac{(s_{21}, s_{02}, s_{10})}{|(s_{21}, s_{02}, s_{10})|} = \frac{(r_{21} - r_{12}, r_{02} - r_{20}, r_{10} - r_{01})}{|(r_{21} - r_{12}, r_{02} - r_{20}, r_{10} - r_{01})|}$$

If $\theta = \pi$, then $R = R^T$ but this time the angle is not zero, so there is a rotation. In this case, notice that

$$R = I + 2S^2 = \begin{bmatrix} 1 - 2(w_1^2 + w_2^2) & 2w_0w_1 & 2w_0w_2 \\ 2w_0w_1 & 1 - 2(w_0^2 + w_2^2) & 2w_1w_2 \\ 2w_0w_2 & 2w_1w_2 & 1 - 2(w_0^2 + w_1^2) \end{bmatrix}$$

where $\mathbf{W} = (w_0, w_1, w_2)$. The idea is to extract the maximum component of the axis from the diagonal entries of the rotation matrix. If r_{00} is maximum, then w_0 must be the largest component in magnitude. Compute $4w_0^2 = r_{00} - r_{11} - r_{22} + 1$ and select $w_0 = \sqrt{r_{00} - r_{11} - r_{22} + 1}/2$. Consequently, $w_1 = r_{01}/(2w_0)$ and $w_2 = r_{02}/(2w_0)$. If r_{11} is maximum, then compute $4w_1^2 = r_{11} - r_{00} - r_{22} + 1$ and select $w_1 = \sqrt{r_{11} - r_{00} - r_{22} + 1}/2$. Consequently, $w_0 = r_{01}/(2w_1)$ and $w_2 = r_{12}/(2w_1)$. Finally, if r_{22} is maximum, then compute $4w_2^2 = r_{22} - r_{00} - r_{11} + 1$ and select $w_2 = \sqrt{r_{22} - r_{00} - r_{11} + 1}/2$. Consequently, $w_0 = r_{02}/(2w_2)$ and $w_1 = r_{12}/(2w_2)$.

17.1.3 INTERPOLATION

The absence of a meaningful interpolation formula that directly applies to rotation matrices is used as an argument for the superiority of quaternions over rotation matrices. However, the *spherical linear interpolation* (or *slerp*) that applies to quaternions has an equivalent formulation for rotation matrices. If P and Q are rotation matrices, then the slerp of the matrices is

$$\text{slerp}(t; P, Q) = P(P^T Q)^t$$

where $t \in [0, 1]$. The technical problem is to define what is meant by R^t for a rotation R and real-valued t . If the rotation has axis direction $\mathbf{W}$ and angle θ , then R^t has the same rotation axis direction, but the angle of rotation is $t\theta$. The procedure for computing the slerp of the rotation matrices is as follows

1. Compute $R = P^T Q$.
2. Extract an axis $\mathbf{W}$ and an angle θ from R .
3. Compute R^t by converting the axis-angle pair $\mathbf{W}$ and $t\theta$ to a rotation matrix.
4. Compute the result $\text{slerp}(t; P, Q) = P R^t$.

17.2 QUATERNIONS

If you have already explored the concept of quaternions, you will have discovered a number of ways to motivate how they are defined. Our interest, of course, is how they are related to rotations. Some of the definitions for quaternions do not make it immediately clear what this relationship is. This section motivates quaternions by looking at 2D rotations but in the space of 4-tuples (x, y, z, w) .

Equation (17.2) is a natural extension of the concept of rotation in the 2D xy -plane, defined in Equation (17.1), to the concept of rotation in the xy -plane that lives in a 3D space. The extension to a rotation in the xy -plane that lives in a 4D space is shown in the following equation:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = R_4 \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \quad (17.10)$$

where the last equality defines the rotation matrix R_4 .

Equation (17.3) showed us how to use the xy -rotation matrix R_3 to build a general 3D rotation matrix, but we had to rely on constructing the identities in Equations (17.4), (17.6), and (17.7) to obtain a matrix that depended only on the rotation angle θ and the rotation axis direction $\mathbf{W}$. The last two identities are inherently 3D because of the use of the cross product operator that is defined for 3D vectors. We can attempt to construct a matrix as shown in Equation (17.3), but first we need to represent the vectors in homogeneous coordinates by setting the w -components to zero. We also need a fourth vector that is perpendicular to the three already given and for which the four vectors form an orthonormal set. Specifically, define

$$\hat{\mathbf{U}} = \begin{bmatrix} \mathbf{U} \\ 0 \end{bmatrix}, \quad \hat{\mathbf{V}} = \begin{bmatrix} \mathbf{V} \\ 0 \end{bmatrix}, \quad \hat{\mathbf{W}} = \begin{bmatrix} \mathbf{W} \\ 0 \end{bmatrix}, \quad \hat{\mathbf{L}} = \begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix}$$

The general rotation matrix involving these is

$$\begin{aligned} R &= [\hat{\mathbf{U}} \quad \hat{\mathbf{V}} \quad \hat{\mathbf{W}} \quad \hat{\mathbf{L}}] \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{U}}^T \\ \hat{\mathbf{V}}^T \\ \hat{\mathbf{W}}^T \\ \hat{\mathbf{L}}^T \end{bmatrix} \\ &= (\cos \theta) (\hat{\mathbf{U}}\hat{\mathbf{U}}^T + \hat{\mathbf{V}}\hat{\mathbf{V}}^T) + (\sin \theta) (\hat{\mathbf{V}}\hat{\mathbf{U}}^T - \hat{\mathbf{U}}\hat{\mathbf{V}}^T) + \hat{\mathbf{W}}\hat{\mathbf{W}}^T + \hat{\mathbf{L}}\hat{\mathbf{L}}^T \end{aligned} \quad (17.11)$$

The construction that led to Equation (17.4) applies equally as well in four dimensions, producing the identity

$$I_4 = \hat{\mathbf{U}}\hat{\mathbf{U}}^T + \hat{\mathbf{V}}\hat{\mathbf{V}}^T + \hat{\mathbf{W}}\hat{\mathbf{W}}^T + \hat{\mathbf{L}}\hat{\mathbf{L}}^T \quad (17.12)$$

where I_4 is the 4×4 identity matrix. The extension of Equation (17.6) to four dimensions is

$$\begin{aligned}
 \hat{\mathbf{v}}\hat{\mathbf{u}}^T - \hat{\mathbf{u}}\hat{\mathbf{v}}^T &= \begin{bmatrix} \hat{\mathbf{v}} \\ 0 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{u}}^T & | & 0 \end{bmatrix} - \begin{bmatrix} \hat{\mathbf{u}} \\ 0 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{v}}^T & | & 0 \end{bmatrix} \\
 &= \begin{bmatrix} \mathbf{v}\mathbf{u}^T - \mathbf{u}\mathbf{v}^T & | & \mathbf{0} \\ \mathbf{0}^T & | & 0 \end{bmatrix} \\
 &= \begin{bmatrix} S & | & \mathbf{0} \\ \mathbf{0}^T & | & 0 \end{bmatrix}
 \end{aligned} \tag{17.13}$$

where S is the 3×3 skew-symmetric matrix defined in Equation (17.5). The extension of Equation (17.7) to four dimensions is

$$\begin{aligned}
 \hat{\mathbf{u}}\hat{\mathbf{u}}^T + \hat{\mathbf{v}}\hat{\mathbf{v}}^T &= \begin{bmatrix} \hat{\mathbf{u}} \\ 0 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{u}}^T & | & 0 \end{bmatrix} + \begin{bmatrix} \hat{\mathbf{v}} \\ 0 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{v}}^T & | & 0 \end{bmatrix} \\
 &= \begin{bmatrix} \mathbf{u}\mathbf{u}^T + \mathbf{v}\mathbf{v}^T & | & \mathbf{0} \\ \mathbf{0}^T & | & 0 \end{bmatrix} \\
 &= \begin{bmatrix} -S^2 & | & \mathbf{0} \\ \mathbf{0}^T & | & 0 \end{bmatrix}
 \end{aligned} \tag{17.14}$$

Replacing Equations (17.12), (17.13), and (17.14) into Equation (17.11) leads to

$$\begin{aligned}
 R &= (\cos \theta) (\hat{\mathbf{u}}\hat{\mathbf{u}}^T + \hat{\mathbf{v}}\hat{\mathbf{v}}^T) + (\sin \theta) (\hat{\mathbf{v}}\hat{\mathbf{u}}^T - \hat{\mathbf{u}}\hat{\mathbf{v}}^T) + \hat{\mathbf{w}}\hat{\mathbf{w}}^T + \hat{\mathbf{l}}\hat{\mathbf{l}}^T \\
 &= (\cos \theta) \begin{bmatrix} -S^2 & | & \mathbf{0} \\ \mathbf{0}^T & | & 0 \end{bmatrix} + (\sin \theta) \begin{bmatrix} S & | & \mathbf{0} \\ \mathbf{0}^T & | & 0 \end{bmatrix} + \begin{bmatrix} I_3 + S^2 & | & \mathbf{0} \\ \mathbf{0}^T & | & 0 \end{bmatrix} \\
 &= \begin{bmatrix} I_3 + (\sin \theta)S + (1 - \cos \theta)S^2 & | & \mathbf{0} \\ \mathbf{0}^T & | & 1 \end{bmatrix} \\
 &= \begin{bmatrix} R_3 & | & \mathbf{0} \\ \mathbf{0}^T & | & 1 \end{bmatrix}
 \end{aligned} \tag{17.15}$$

where I_3 is the 3×3 identity matrix.

Equation (17.15) is no surprise and is, in fact, disappointing. Embedding the 3D rotation into a 4D space appears to have given us no new insight into rotations. However, there is a clever way of handling rotations so that we can obtain information from the 4D approach.

17.2.1 THE LINEAR ALGEBRAIC VIEW OF QUATERNIONS

Equation (17.10) represents a rotation in the (x, y) -components of the 4-tuple (x, y, z, w) and the (z, w) -components are unchanged. It is possible to rotate twice using half the rotation angle per application. In block-matrix form,

$$R_4 = \left[\begin{array}{c|c} H_2 & Z_2 \\ \hline Z_2^T & I_2 \end{array} \right] \left[\begin{array}{c|c} H_2 & Z_2 \\ \hline Z_2^T & I_2 \end{array} \right] = \left[\begin{array}{c|c} R_2 & Z_2 \\ \hline Z_2^T & I_2 \end{array} \right]$$

where Z_2 is the 2×2 zero matrix, I_2 is the 2×2 identity matrix, and

$$H_2 = \begin{bmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{bmatrix}$$

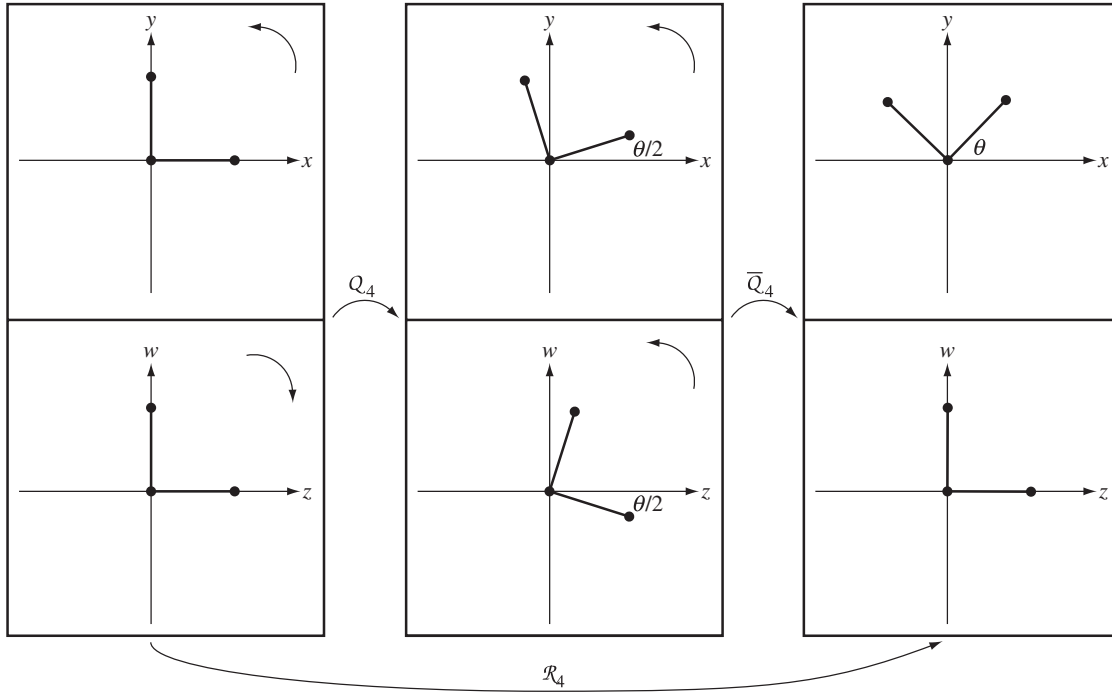
Each half-angle rotation applies to the (x, y) -components, but the (z, w) -components never change. The observation that allows us to gain some insight in the 4D approach is that we need not use I_2 in the lower-right corner of the matrices. The first I_2 may be replaced by an invertible matrix M and the second I_2 may be replaced by the inverse M^{-1} . Thus, the (z, w) -components change to (z', w') by the first matrix but then (z', w') changes back to (z, w) by the second matrix. In fact, we will use H_2 and its inverse for the (z, w) -components:

$$R_4 = \left[\begin{array}{c|c} H_2 & Z_2 \\ \hline Z_2^T & H_2 \end{array} \right] \left[\begin{array}{c|c} H_2 & Z_2 \\ \hline Z_2^T & H_2^T \end{array} \right] = \overline{Q}_4 Q_4 \quad (17.16)$$

where the last equality defines the matrix Q_4 and $\overline{Q}_4$, themselves rotations in four dimensions. Figure 17.1 illustrates the application of the two half-angle rotations.

In summary, the half-angle rotation H_2 is applied twice to (x, y) to obtain the full-angle rotation in the xy -plane. The inverse half-angle rotation H_2^T is applied to (z, w) , a rotation within the zw -plane; however, that rotation is undone by H in the second operation, the end result being that (z, w) is unchanged by the composition.

Now we may attempt to construct a general rotation, using the ideas that went into Equation (17.11) but applied to each of the half-angle rotation matrices Q_4 and $\overline{Q}_4$. The resulting matrices are named Q and $\overline{Q}$.

Figure 17.1 A 3D rotation about the z -axis that is represented as the product of two 4D rotations.

$$\begin{aligned}
 Q &= [\hat{\mathbf{U}} \quad \hat{\mathbf{V}} \quad \hat{\mathbf{W}} \quad \hat{\mathbf{L}}] \begin{bmatrix} \cos(\theta/2) & -\sin(\theta/2) & 0 & 0 \\ \sin(\theta/2) & \cos(\theta/2) & 0 & 0 \\ 0 & 0 & \cos(\theta/2) & \sin(\theta/2) \\ 0 & 0 & -\sin(\theta/2) & \cos(\theta/2) \end{bmatrix} \begin{bmatrix} \hat{\mathbf{U}}^T \\ \hat{\mathbf{V}}^T \\ \hat{\mathbf{W}}^T \\ \hat{\mathbf{L}}^T \end{bmatrix} \\
 &= (\cos(\theta/2)) (\hat{\mathbf{U}}\hat{\mathbf{U}}^T + \hat{\mathbf{V}}\hat{\mathbf{V}}^T + \hat{\mathbf{W}}\hat{\mathbf{W}}^T + \hat{\mathbf{L}}\hat{\mathbf{L}}^T) \\
 &\quad + (\sin(\theta/2)) (\hat{\mathbf{V}}\hat{\mathbf{U}}^T - \hat{\mathbf{U}}\hat{\mathbf{V}}^T + \hat{\mathbf{W}}\hat{\mathbf{L}}^T - \hat{\mathbf{L}}\hat{\mathbf{W}}^T)
 \end{aligned} \tag{17.17}$$

We already have in place the identities of Equations (17.12) and (17.13). Another identity is

$$\hat{\mathbf{W}}\hat{\mathbf{L}}^T - \hat{\mathbf{L}}\hat{\mathbf{W}}^T = \begin{bmatrix} Z_3 & \mathbf{W} \\ -\mathbf{W} & 0 \end{bmatrix} \tag{17.18}$$

where Z_3 is the 3×3 zero matrix. Substituting these identities into Equation (17.17) produces

$$Q = \begin{bmatrix} \cos(\theta/2)I_3 + \sin(\theta/2)S & \sin(\theta/2)\mathbf{W} \\ -\sin(\theta/2)\mathbf{W}^T & \cos(\theta/2) \end{bmatrix} \quad (17.19)$$

where I_3 is the 3×3 identity matrix and S is the skew-symmetric matrix of Equation (17.5). Observe that Equation (17.19) depends only on the rotation angle θ and the rotation axis direction $\mathbf{W}$; the matrix S depends only on the components of $\mathbf{W}$. A similar construction leads to

$$\overline{Q} = \begin{bmatrix} \cos(\theta/2)I_3 + \sin(\theta/2)S & -\sin(\theta/2)\mathbf{W} \\ \sin(\theta/2)\mathbf{W}^T & \cos(\theta/2) \end{bmatrix} \quad (17.20)$$

We may combine all the calculations here to produce the 4×4 matrix that corresponds to the general rotation whose 3×3 matrix is R of Equation (17.3).

$$\begin{aligned} R &= [\hat{\mathbf{U}} \quad \hat{\mathbf{V}} \quad \hat{\mathbf{W}} \quad \hat{\mathbf{L}}] \overline{Q}_4 Q_4 \begin{bmatrix} \hat{\mathbf{U}}^T \\ \hat{\mathbf{V}}^T \\ \hat{\mathbf{W}}^T \\ \hat{\mathbf{L}}^T \end{bmatrix} \\ &= [\hat{\mathbf{U}} \quad \hat{\mathbf{V}} \quad \hat{\mathbf{W}} \quad \hat{\mathbf{L}}] \overline{Q}_4 I_4 Q_4 \begin{bmatrix} \hat{\mathbf{U}}^T \\ \hat{\mathbf{V}}^T \\ \hat{\mathbf{W}}^T \\ \hat{\mathbf{L}}^T \end{bmatrix} \\ &= [\hat{\mathbf{U}} \quad \hat{\mathbf{V}} \quad \hat{\mathbf{W}} \quad \hat{\mathbf{L}}] \overline{Q}_4 \begin{bmatrix} \hat{\mathbf{U}}^T \\ \hat{\mathbf{V}}^T \\ \hat{\mathbf{W}}^T \\ \hat{\mathbf{L}}^T \end{bmatrix} [\hat{\mathbf{U}} \quad \hat{\mathbf{V}} \quad \hat{\mathbf{W}} \quad \hat{\mathbf{L}}] Q_4 \begin{bmatrix} \hat{\mathbf{U}}^T \\ \hat{\mathbf{V}}^T \\ \hat{\mathbf{W}}^T \\ \hat{\mathbf{L}}^T \end{bmatrix} \quad (17.21) \\ &= \overline{Q} Q \\ &= \begin{bmatrix} \cos(\theta/2)I_3 + \sin(\theta/2)S & -\sin(\theta/2)\mathbf{W} \\ \sin(\theta/2)\mathbf{W}^T & \cos(\theta/2) \end{bmatrix} \begin{bmatrix} \cos(\theta/2)I_3 + \sin(\theta/2)S & \sin(\theta/2)\mathbf{W} \\ -\sin(\theta/2)\mathbf{W}^T & \cos(\theta/2) \end{bmatrix} \\ &= \begin{bmatrix} I_3 + (\sin \theta)S + (1 - \cos \theta)S^2 & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \end{aligned}$$

The matrix Q in Equation (17.19) may be expanded to produce the following, where $\mathbf{W} = (w_0, w_1, w_2)$, $\sigma = \sin(\theta/2)$, and $\gamma = \cos(\theta/2)$:

$$Q = \begin{bmatrix} \gamma & -\sigma w_2 & \sigma w_1 & \sigma w_0 \\ \sigma w_2 & \gamma & -\sigma w_0 & \sigma w_1 \\ -\sigma w_1 & \sigma w_0 & \gamma & \sigma w_2 \\ -\sigma w_0 & -\sigma w_1 & -\sigma w_2 & \gamma \end{bmatrix} \quad (17.22)$$

Although Q has 16 entries, only four of them are unique—the last column values. Moreover, $\overline{Q}$ of Equation (17.20) uses these same four values:

$$\overline{Q} = \begin{bmatrix} \gamma & -\sigma w_2 & \sigma w_1 & -\sigma w_0 \\ \sigma w_2 & \gamma & -\sigma w_0 & -\sigma w_1 \\ -\sigma w_1 & \sigma w_0 & \gamma & -\sigma w_2 \\ \sigma w_0 & \sigma w_1 & \sigma w_2 & \gamma \end{bmatrix} \quad (17.23)$$

17.2.2 ROTATION OF A VECTOR

The rotation of a vector (x, y, z) using the 4D representation of Equation (17.21) is accomplished by appending a zero component to the vector:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 0 \end{bmatrix} R \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \overline{Q} Q \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} \quad (17.24)$$

An implementation will take advantage of the fact that the input vector has a zero component, in which case it is not necessary to multiply matrix entries by that component. The product of Q and the input generally has four nonzero components, but since you know the output has a last component of zero, you need not multiply the last row of $\overline{Q}$ with the intermediate vector obtained after multiplication by Q . This saves some calculation time. If you are using hardware support for matrix-vector multiplication, then this is a moot point.

17.2.3 PRODUCT OF ROTATIONS

Consider two 3D rotations represented by 4D matrices, say, $R_0 = \overline{Q}_0 Q_0$ and $R_1 = \overline{Q}_1 Q_1$. The product is

$$\begin{aligned} R_1 R_0 &= (\overline{Q}_1 Q_1)(\overline{Q}_0 Q_0) = \overline{Q}_1 (Q_1 \overline{Q}_0) Q_0 \\ &= \overline{Q}_1 (\overline{Q}_0 Q_1) Q_0 = (\overline{Q}_1 \overline{Q}_0) (Q_1 Q_0) \end{aligned} \quad (17.25)$$

The calculations involve associativity of matrix multiplication except for one calculation where the order of two matrices is reversed. Generally, matrix multiplication is not commutative. However, it is true for the calculation used here.

EXERCISE 17.1

Verify that $Q_1 \overline{Q}_0 - \overline{Q}_0 Q_1 = 0$. Use Equations (17.19) and (17.20) to set up the block-matrix equation. Expand the left-hand side using matrix and vector arithmetic and show that all the block terms are zero. ■

The four distinct entries in Q_i are all you need to know about the rotation matrix R_i for $i = 0, 1$. Since $R_1 R_0$ is a rotation matrix, it may also be represented by $R_1 R_0 = \overline{Q} Q$ for some matrix Q with four distinct entries. Equation (17.25) says that $Q = Q_1 Q_0$. For the purpose of composition of rotations, it is enough to store the four distinct entries for each matrix, compute $Q_1 Q_0$, and extract its four distinct entries.

17.2.4 THE CLASSICAL VIEW OF QUATERNIONS

Although the standard way quaternions are introduced is by simply defining them as symbolic quantities and defining their algebraic properties, I will use the discussion from the previous section to introduce quaternions. I find the linear algebraic approach quite intuitive.

We found that a 3D rotation about an axis with direction $\mathbf{W}$ by an angle θ may be represented as a 4D rotation matrix

$$R = \overline{Q} Q$$

where Q and $\overline{Q}$ are matrices defined by Equations (17.22) and (17.23). We also found that applying the rotation to a vector written in homogeneous coordinates $\mathbf{V} = [x \ y \ z \ 0]^T$ is accomplished by

$$R\mathbf{V} = \overline{Q} Q \mathbf{V}$$

Given two rotations represented as $R_0 = \overline{Q}_0 Q_0$ and $R_1 = \overline{Q}_1 Q_1$, the composition is

$$R_0 R_1 = (\overline{Q}_0 \overline{Q}_1)(Q_0 Q_1)$$

It is necessary, therefore, to keep track only of the Q parts of the rotations, using them for multiplication of vectors or for composition of matrices.

Let us examine Q in more detail. Equation (17.22) may be expanded as shown.

$$\begin{aligned} Q &= \gamma \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} + \sigma w_0 \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix} \\ &+ \sigma w_1 \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix} + \sigma w_2 \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (17.26) \\ &= \cos(\theta/2) \mathbf{1} + \sin(\theta/2) (w_0 \mathbf{i} + w_1 \mathbf{j} + w_2 \mathbf{k}) \end{aligned}$$

Table 17.1 Products of the four matrices $\{\mathbf{1}, \mathbf{i}, \mathbf{j}, \mathbf{k}\}$. If $\mathbf{A}$ is a matrix from a row of the table and if $\mathbf{B}$ is a matrix from a column of the table, the table entry in that row and column is $\mathbf{AB}$.

	1	i	j	k
1	1	i	j	k
i	i	-1	k	$-j$
j	j	$-k$	-1	i
k	k	j	$-i$	-1

where the last equality defines the four 4×4 matrices **1**, **i**, **j**, and **k**. A boldface font is used here for the matrices, so be careful not to confuse the matrix **1** with the scalar 1. Equation (17.26) is a compact representation of a rotation, involving only the rotation angle θ and the components of the rotation axis direction $\mathbf{W} = (w_0, w_1, w_2)$.

If you have seen quaternions before, Equation (17.26) should look familiar. A (unit-length) quaternion is normally written in the classical form as

$$q = \cos(\theta/2) + \sin(\theta/2)(w_0i + w_1j + w_2k) = \cos(\theta/2) + \sin(\theta/2)\hat{w} \quad (17.27)$$

where the last equality defines $\hat{w}$. Its relationship to 3D rotation is exactly what has been discussed in the previous material. You will also have seen definitions for multiplication of quaternions. The i -, j -, and k -terms appear to be similar to the imaginary part of a complex-valued number, but with a richer structure. The products are $i^2 = -1$, $j^2 = -1$, $k^2 = -1$, $ij = k$, $ji = -k$, $ki = j$, $ik = -j$, $jk = i$, and $kj = -i$. If you instead consider the matrix representation of Equation (17.26) and think of **1**, **i**, **j**, and **k** as the matrix representations of 1, i , j , and k , respectively, then the matrix products produce the same results. Table 17.1 lists the products of the four matrices.

The matrix $\overline{Q}$ has a representation similar to that of Q , namely,

$$\overline{Q} = \cos(\theta/2)\mathbf{1} - \sin(\theta/2)(w_0\mathbf{i} + w_1\mathbf{j} + w_2\mathbf{k}) \quad (17.28)$$

The quaternion representation is

$$\bar{q} = \cos(\theta/2) - \sin(\theta/2)(w_0i + w_1j + w_2k) = \cos(\theta/2) - \sin(\theta/2)\hat{w} \quad (17.29)$$

and is said to be the *conjugate* of q .

The rotation of a vector using the 4D matrix approach is $\mathbf{U} = \overline{Q}Q\mathbf{V}$, where as 4-tuples, $\mathbf{U} = (u_0, u_1, u_2, 0)$ and $\mathbf{V} = (v_0, v_1, v_2, 0)$. Let q be the quaternion represented by Q and let $\bar{q}$ be the quaternion represented by $\overline{Q}$. Let $\hat{v} = v_0i + v_1j + v_2k$

be a quaternion representing $\mathbf{V}$. The rotated vector is produced by the quaternion product

$$\hat{u} = q\hat{v}\bar{q} \quad (17.30)$$

where $\hat{u} = u_0i + u_1j + u_2k$.

In terms of 4D matrices, the composition of rotation matrices $R_0 = \bar{Q}_0Q_0$ and $R_1 = \bar{Q}_1Q_1$ was shown to be $R_0R_1 = (\bar{Q}_0\bar{Q}_1)(Q_0Q_1)$, so it is sufficient to compute the four unique entries in Q_0Q_1 . If $\sigma_i = \sin(\theta_i/2)$, $\gamma_i = \cos(\theta_i/2)$, and $\mathbf{W}_i$ is the axis direction with corresponding skew-symmetric matrices S_i for $i = 0, 1$, then a direct computation of the product is

$$\begin{aligned} Q_0Q_1 &= \left[\begin{array}{c|c} \gamma_0I + \sigma_0S_0 & \sigma_0\mathbf{W}_0 \\ \hline -\sigma_0\mathbf{W}_0^T & \gamma_0 \end{array} \right] \left[\begin{array}{c|c} \gamma_1I + \sigma_1S_1 & \sigma_1\mathbf{W}_1 \\ \hline -\sigma_1\mathbf{W}_1^T & \gamma_1 \end{array} \right] \\ &= \left[\begin{array}{c|c} (\gamma_0I + \sigma_0S_0)(\gamma_1I + \sigma_1S_1) - \sigma_0\sigma_1\mathbf{W}_0\mathbf{W}_1^T & \sigma_0\gamma_1\mathbf{W}_0 + \sigma_1\gamma_0\mathbf{W}_1 + \sigma_0\sigma_1S_0\mathbf{W}_1 \\ \hline -\sigma_0\gamma_1\mathbf{W}_0^T - \sigma_1\gamma_0\mathbf{W}_1^T - \sigma_0\sigma_1\mathbf{W}_0^TS_1 & \gamma_0\gamma_1 - \sigma_0\sigma_1\mathbf{W}_0^T\mathbf{W}_1 \end{array} \right] \end{aligned}$$

The rightmost column lists the unique entries of the product. In terms of quaternions q_0 and q_1 represented by Q_0 and Q_1 ,

$$q_0q_1 = (\gamma_0\gamma_1 - \sigma_0\sigma_1\hat{w}_0 \cdot \hat{w}_1) + \sigma_0\gamma_1\hat{w}_0 + \sigma_1\gamma_0\hat{w}_1 + \sigma_0\sigma_1\hat{w}_0 \times \hat{w}_1 \quad (17.31)$$

where $\hat{w}_0$ and $\hat{w}_1$ are quaternions representing $\mathbf{W}_0$ and $\mathbf{W}_1$, each having only i -, j -, and k -components but a zero component for the 1-term. We know that $S_0\mathbf{W}_1 = \mathbf{W}_0 \times \mathbf{W}_1$, so $\hat{w}_0 \times \hat{w}_1$ is the quaternion that represents the cross product of vectors.

17.2.5 AXIS/ANGLE TO QUATERNION

If $\mathbf{W} = (w_0, w_1, w_2)$ is the direction of the rotation axis and if θ is the rotation angle, then a corresponding quaternion is

$$q = \cos(\theta/2) + \sin(\theta/2)(w_0i + w_1j + w_2k)$$

The unit quaternions provide a *double covering* of the rotations. If q represents the rotation, then so does

$$\begin{aligned} -q &= -\cos(\theta/2) - \sin(\theta/2)(w_0i + w_1j + w_2k) \\ &= \cos((\theta + 2\pi)/2) + \sin((\theta + 2\pi)/2)(w_0i + w_1j + w_2k) \end{aligned}$$

As you can see, q represents rotation about the axis by angle θ and $-q$ represents rotation about the axis by angle $\theta + 2\pi$, but these are the same rotation when applied to vectors.

17.2.6 QUATERNION TO AXIS/ANGLE

Let $q = w + xi + yj + zk$ be a unit quaternion. If $|w| = 1$, then the angle is $\theta = 0$ and any unit-length direction vector for the axis will do since there is no rotation. If $|w| \neq 1$, the angle is obtained as $\theta = 2 \cos^{-1}(w)$ and the axis direction is computed as $\mathbf{W} = (x, y, z)/\sqrt{1 - w^2}$.

17.2.7 MATRIX TO QUATERNION

The 3D rotation matrix is R and the corresponding quaternion is $q = w + xi + yj + zk$. Previously, it was mentioned that $\cos \theta = (\text{Trace}(R) - 1)/2$. Using the identity $2 \cos^2(\theta/2) = 1 + \cos \theta$ yields $w^2 = \cos^2(\theta/2) = (\text{Trace}(R) + 1)/4$ or $|w| = \sqrt{\text{Trace}(R) + 1}/2$. If $\text{Trace}(R) > 0$, then $|w| > 1/2$, so without loss of generality, choose w to be the positive square root, $w = \sqrt{\text{Trace}(R) + 1}/2$. The identity $R - R^T = (2 \sin \theta)S$ also yielded $(r_{12} - r_{21}, r_{20} - r_{02}, r_{01} - r_{10}) = 2 \sin \theta(w_0, w_1, w_2)$. Finally, it is easily shown that $2xw = w_0 \sin \theta$, $2yw = w_1 \sin \theta$, and $2zw = w_2 \sin \theta$. Combining these leads to $x = (r_{12} - r_{21})/(4w)$, $y = (r_{20} - r_{02})/(4w)$, and $z = (r_{01} - r_{10})/(4w)$.

If $\text{Trace}(R) \leq 0$, then $|w| \leq 1/2$. The idea is to first extract the largest one of x , y , or z from the diagonal terms of the rotation R in Equation (17.8). If r_{00} is the maximum diagonal term, then x is larger in magnitude than y or z . Some algebra shows that $4x^2 = r_{00} - r_{11} - r_{22} + 1$ from which is chosen $x = \sqrt{r_{00} - r_{11} - r_{22} + 1}/2$. Consequently, $w = (r_{12} - r_{21})/(4x)$, $y = (r_{01} + r_{10})/(4x)$, and $z = (r_{02} + r_{20})/(4x)$. If r_{11} is the maximum diagonal term, then compute $4y^2 = r_{11} - r_{00} - r_{22} + 1$ and choose $y = \sqrt{r_{11} - r_{00} - r_{22} + 1}/2$. Consequently, $w = (r_{20} - r_{02})/(4y)$, $x = (r_{01} + r_{10})/(4y)$, and $z = (r_{12} + r_{21})/(4y)$. Finally, if r_{22} is the maximum diagonal term, then compute $4z^2 = r_{22} - r_{00} - r_{11} + 1$ and choose $z = \sqrt{r_{22} - r_{00} - r_{11} + 1}/2$. Consequently, $w = (r_{01} - r_{10})/(4z)$, $x = (r_{02} + r_{20})/(4z)$, and $y = (r_{12} + r_{21})/(4z)$.

17.2.8 QUATERNION TO MATRIX

The quaternion is $q = w + xi + yj + zk$ and the 3D rotation matrix is R . Using the identities $2 \sin^2(\theta/2) = 1 - \cos(\theta)$ and $\sin(\theta) = 2 \sin(\theta/2) \cos(\theta/2)$, it is easily shown that $2wx = (\sin \theta)w_0$, $2wy = (\sin \theta)w_1$, $2wz = (\sin \theta)w_2$, $2x^2 = (1 - \cos \theta)w_0^2$, $2xy = (1 - \cos \theta)w_0w_1$, $2xz = (1 - \cos \theta)w_0w_2$, $2y^2 = (1 - \cos \theta)w_1^2$, $2yz = (1 - \cos \theta)w_1w_2$, and $2z^2 = (1 - \cos \theta)w_2^2$. The right-hand sides of all these equations are terms in the expression $R = I + (\sin \theta)S + (1 - \cos \theta)S^2$. Replacing them yields

$$R = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2wz & 2xz + 2wy \\ 2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx \\ 2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 \end{bmatrix}$$

17.2.9 INTERPOLATION

Quaternions are quite amenable to interpolation. Once again, the standard operation that is used is *spherical linear interpolation* (*slerp*). Given quaternions p and q with acute angle θ between them, *slerp* is defined as

$$\text{slerp}(t; p, q) = p(p^*q)^t$$

for $t \in [0, 1]$. Note that $\text{slerp}(0; p, q) = p$ and $\text{slerp}(1; p, q) = q$. It is not immediately clear how to compute *slerp* in the form specified by the definition. An equivalent definition is

$$\text{slerp}(t; p, q) = \frac{\sin((1-t)\theta)p + \sin(t\theta)q}{\sin(\theta)}$$

If p and q are thought of as points on a unit circle, this formula is a parameterization of the shortest arc between them. If a particle travels on that curve according to the parameterization, it does so with constant speed. Thus, any uniform sampling of t in $[0, 1]$ produces equally spaced points on the arc.

We assume that only p , q , and t are specified. Moreover, since q and $-q$ represent the same rotation, you can replace q by $-q$, if necessary, to guarantee that the angle between p and q treated as 4-tuples is acute. That is, $p \cdot q \geq 0$. As 4-tuples, p and q are unit length. The dot product is therefore $p \cdot q = \cos(\theta)$.

17.3 EULER ANGLES

Rotations about the coordinate axes are easy to define and work with. Rotation about the x -axis by angle θ is

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}$$

where $\theta > 0$ indicates a counterclockwise rotation in the plane $x = 0$. The observer is assumed to be positioned on the side of the plane with $x > 0$ and looking at the origin. Rotation about the y -axis by angle θ is

$$R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix}$$

where $\theta > 0$ indicates a counterclockwise rotation in the plane $y = 0$. The observer is assumed to be positioned on the side of the plane with $y > 0$ and looking at the

origin. Rotation about the z -axis by angle θ is

$$R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where $\theta > 0$ indicates a counterclockwise rotation in the plane $z = 0$. The observer is assumed to be positioned on the side of the plane with $z > 0$ and looking at the origin. Rotation by an angle θ about an arbitrary axis containing the origin and having unit-length direction $\mathbf{W} = (w_0, w_1, w_2)$ is given by

$$R_{\mathbf{W}}(\theta) = I + (\sin \theta)S + (1 - \cos \theta)S^2$$

where I is the identity matrix,

$$S = \begin{bmatrix} 0 & -w_2 & w_1 \\ w_2 & 0 & -w_0 \\ -w_1 & w_0 & 0 \end{bmatrix}$$

and $\theta > 0$ indicates a counterclockwise rotation in the plane through the origin and perpendicular to $\mathbf{W}$. The observer is assumed to be positioned on the side of the plane to which $\mathbf{W}$ points and is looking at the origin.

A common problem is to want a factorization of a rotation matrix R as a product of rotations about the coordinate axes. The form of the factorization depends on the needs of the application and what ordering is specified. For example, you might want to factor the rotation as

$$R = R_x(\theta_x)R_y(\theta_y)R_z(\theta_z)$$

for some angles θ_x , θ_y , and θ_z , which are called *Euler angles*. The ordering is xyz . Five other possibilities are xzy , yxz , yzx , zxy , and zyx . Also, it is not required that there be a rotation about each of the coordinate axes; factorizations such as xyx are allowed:

$$R = R_x(\phi_x)R_y(\theta_y)R_x(\theta_x)$$

All the possibilities so far involve specifying rotations about the original coordinate axes. It is possible to use rotations about the rotated axes. For example,

$$R = R_{\mathbf{U}_0}(\theta_0)R_{\mathbf{U}_1}(\theta_1)R_{\mathbf{U}_2}(\theta_2)$$

where $\mathbf{U}_2$ is the last column of the identity matrix; $\mathbf{U}_1$ is the middle column of the matrix $R_{\mathbf{U}_2}(\theta_2)$; and $\mathbf{U}_0$ is the first column of the matrix $R_{\mathbf{U}_1}(\theta_1)R_{\mathbf{U}_2}(\theta_2)$.

In all cases, you can formulate an algorithm for the factorization. To illustrate, consider $R = R_x(\theta_x)R_y(\theta_y)R_z(\theta_z)$. Setting $R = [r_{ij}]$ for $0 \leq i \leq 2$ and $0 \leq j \leq 2$,

formally multiplying $R_x(\theta_x)R_y(\theta_y)R_z(\theta_z)$, and equating yields

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} c_y c_z & -c_y s_z & s_y \\ c_z s_x s_y + c_x s_z & c_x c_z - s_x s_y s_z & -c_y s_x \\ -c_x c_z s_y + s_x s_z & c_z s_x + c_x s_y s_z & c_x c_y \end{bmatrix}$$

where $c_a = \cos(\theta_a)$ and $s_a = \sin(\theta_a)$ for a equal to x , y , or z . From this we have $s_y = r_{02}$, so $\theta_y = \sin^{-1}(r_{02})$.

If $\theta_y \in (-\pi/2, \pi/2)$, then $c_y \neq 0$ and $c_y(s_x, c_x) = (-r_{12}, r_{22})$, in which case $\theta_x = \text{atan2}(-r_{12}, r_{22})$. Similarly, $c_y(s_z, c_z) = (-r_{01}, r_{00})$, in which case $\theta_z = \text{atan2}(-r_{01}, r_{00})$.

If $\theta_y = \pi/2$, then $s_y = 1$ and $c_y = 0$. In this case,

$$\begin{bmatrix} r_{10} & r_{11} \\ r_{20} & r_{21} \end{bmatrix} = \begin{bmatrix} c_z s_x + c_x s_z & c_x c_z - s_x s_z \\ -c_x c_z + s_x s_z & c_z s_x + c_x s_z \end{bmatrix} = \begin{bmatrix} \sin(\theta_z + \theta_x) & \cos(\theta_z + \theta_x) \\ -\cos(\theta_z + \theta_x) & \sin(\theta_z + \theta_x) \end{bmatrix}.$$

Therefore, $\theta_z + \theta_x = \text{atan2}(r_{10}, r_{11})$. There is one degree of freedom, so the factorization is not unique. One choice is $\theta_z = 0$ and $\theta_x = \text{atan2}(r_{10}, r_{11})$.

If $\theta_y = -\pi/2$, then $s_y = -1$ and $c_y = 0$. In this case,

$$\begin{bmatrix} r_{10} & r_{11} \\ r_{20} & r_{21} \end{bmatrix} = \begin{bmatrix} -c_z s_x + c_x s_z & c_x c_z + s_x s_z \\ c_x c_z + s_x s_z & c_z s_x - c_x s_z \end{bmatrix} = \begin{bmatrix} \sin(\theta_z - \theta_x) & \cos(\theta_z - \theta_x) \\ \cos(\theta_z - \theta_x) & -\sin(\theta_z - \theta_x) \end{bmatrix}.$$

Therefore, $\theta_z - \theta_x = \text{atan2}(r_{10}, r_{11})$. There is one degree of freedom, so the factorization is not unique. One choice is $\theta_z = 0$ and $\theta_x = -\text{atan2}(r_{10}, r_{11})$.

Pseudocode for the factorization is

```
thetaY = asin(r02);
if (thetaY < PI/2)
{
    if (thetaY > -PI/2)
    {
        thetaX = atan2(-r12,r22);
        thetaZ = atan2(-r01,r00);
    }
    else
    {
        // not a unique solution
        thetaX = -atan2(r10,r11);
        thetaZ = 0;
    }
}
else
```

```

{
    // not a unique solution
    thetaX = atan2(r10,r11);
    thetaZ = 0;
}

```

The other combinations such as xzy and xyx may be factored similarly.

17.4 PERFORMANCE ISSUES

A question asked quite often is, What is the best representation to use for rotations: matrices, quaternions, or axis/angle pairs? As with most computer science topics, there is no answer to this question, only trade-offs to consider. You might consider operation counting as a first-order measure of performance, counting multiplications, additions and subtractions, divisions, and function evaluations (such as `sqrt` or `cos`). With current CPU technology, operation counting alone is not a sufficient measure of performance. Memory access patterns can lead to good cache coherence or to cache misses. Branch penalties for conditional statements can be quite expensive. Any amount of parallelism, small or large, can affect the final comparisons of algorithms. Your best bet for comparison is to implement the competing algorithms and time them with reasonable data.

Regarding memory usage, quaternions require four floating-point values but rotation matrices require nine floating-point values—or 16 values if you require 16-byte alignment on consoles or CPUs with SIMD support. If you have a lot of transformation data and memory is a scarce resource, then the memory usage alone argues for using quaternions.

Regarding performance, your experiments should take into account the times for operations such as matrix-vector products, matrix-matrix products, and conversion between representations. Also, one key argument used for choosing quaternions over rotation matrices is that quaternions naturally support spherical linear interpolation. As I showed previously, spherical linear interpolation may also be applied to rotation matrices. I used to argue that `slerp` for quaternions is much faster than `slerp` for rotation matrices. My conclusion was based on operation counting. I have implementations for `slerp` in both the `Matrix3` and `Quaternion` classes. Recently, I timed the two on an Intel Pentium D (dual core), 3.2-GHz processor (using only one core). Surprisingly, at least to me, is that the rotation `slerp` performed better at a speedup of 1.2. This was measured with a Release build using Microsoft Visual Studio .NET 2005.¹ I imagine it is possible to inline all the code for the algorithms and change the results,

1. Just for kicks, I ran the Debug build and saw that the quaternion `slerp` performed better than rotation `slerp` with a speedup of 2.3.

but the point is that the best way to compare performance is to profile on your target platforms rather than rely on an on-the-napkin calculation.

17.5 THE CURSE OF NONUNIFORM SCALING

In any system that uses transformations involving translation, rotation, and scaling, including composition of all of these, it is a natural expectation that you can factor a composition of transformations into its primitive components of translation, rotation, and scale. How to do this is a frequently asked question in Usenet groups and in game developer forums.

The mathematical setup is as follows. Suppose you store your transformations so that they apply to vectors as follows:

$$\mathbf{Y} = R\mathbf{S}\mathbf{X} + \mathbf{T}$$

where $\mathbf{X}$ is the 3×1 input vector; $\mathbf{Y}$ is the 3×1 output vector; R is a 3×3 rotation matrix; S is a scaling matrix (a diagonal matrix whose diagonal entries are positive); and $\mathbf{T}$ is a 3×1 translation vector.

Now suppose you have a composition of two such transformations. The first transformation is

$$\mathbf{Y} = R_0 S_0 \mathbf{X} + \mathbf{T}_0$$

and the second transformation is

$$\mathbf{Z} = R_1 S_1 \mathbf{Y} + \mathbf{T}_1$$

The composition is

$$\mathbf{Z} = R_1 S_1 (R_0 S_0 \mathbf{X} + \mathbf{T}_0) + \mathbf{T}_1 = (R_1 S_1 R_0 S_0) \mathbf{X} + (R_1 S_1 \mathbf{T}_0 + \mathbf{T}_1) = M\mathbf{X} + \mathbf{T}$$

where the last equality defines the matrix

$$M = R_1 S_1 R_0 S_0$$

and the vector

$$\mathbf{T} = R_1 S_1 \mathbf{T}_0 + \mathbf{T}_1$$

The translational component is immediately available in the composition. The hope is to factor

$$M = RS$$

where R is a rotation matrix and S is a scaling matrix, diagonal and with positive diagonal entries. The technical problem, and one many programmers do not expect, is that *it is not always possible to factor M in this manner*.

EXAMPLE
17.1

Consider the attempt to factor the following matrix:

$$M = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = RS = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} d_0 & 0 \\ 0 & d_1 \end{bmatrix} = \begin{bmatrix} d_0 \cos \theta & -d_1 \sin \theta \\ d_0 \sin \theta & d_1 \cos \theta \end{bmatrix}$$

where $d_0 > 0$ and $d_1 > 0$. Equating the off-diagonal terms, we have $0 = d_0 \sin \theta$ and $1 = -d_1 \sin \theta$. Since $d_0 > 0$, the first equation forces $\sin \theta = 0$. The second equation does not allow $\sin \theta = 0$. Even if you allowed $d_0 = 0$, the equation $d_0 \cos \theta = 1$ is not possible. Therefore, it is not possible to factor M into a rotation matrix times a scale matrix.

Think about it geometrically. The matrix M is a *shear matrix*. The y -values are unchanged by the transformation, but the x -values are sheared to $x + y$. Imagine a square with vertices $(0, 0)$, $(1, 0)$, $(1, 1)$, and $(0, 1)$. The shearing causes the square to become a parallelogram with vertices $(0, 0)$, $(1, 0)$, $(2, 1)$, and $(1, 1)$. A factorization $M = RS$ is an attempt to represent the shearing as a scaling along the x - and y -axes followed by a rotation. The scaling causes the square to become a rectangle with vertices $(0, 0)$, $(d_0, 0)$, (d_0, d_1) , and $(0, d_1)$. No amount of rotation will make this look like the parallelogram produced by M . ■

The inability to factor M into a rotation matrix times a scaling matrix has thrown a damper on many developers' transformation systems—where they were hoping to store $\{R, S, T\}$ for the world transformations at each node in a hierarchy of nodes. The best you can do regarding factorization is *polar decomposition* or *singular value decomposition*, both topics briefly discussed here. I also mention a factorization method called *Gram-Schmidt orthonormalization*, which is a common method used to renormalize the columns of a composition of rotation matrices. The numerical round-off errors in the composition can eventually lead to a matrix that is sufficiently different from a rotation matrix that it needs such an adjustment.

17.5.1 GRAM-SCHMIDT ORTHONORMALIZATION

Given three linearly independent vectors $\mathbf{V}_0$, $\mathbf{V}_1$, and $\mathbf{V}_2$, it is possible to construct an orthonormal set of vectors from them, $\{\mathbf{U}_0, \mathbf{U}_1, \mathbf{U}_2\}$, using the process called Gram-Schmidt orthonormalization. In fact, the method applies to a collection of n linearly independent vectors in an n -dimensional space, but for the purposes of this book, the illustration in three dimensions will suffice.

The first vector in the orthonormal set is chosen to be the normalized vector for $\mathbf{V}_0$, namely,

$$\mathbf{U}_0 = \frac{\mathbf{V}_0}{|\mathbf{V}_0|}$$

The second vector is obtained from $\mathbf{V}_1$ by projecting out the $\mathbf{U}_0$ component, thus producing a vector perpendicular to $\mathbf{U}_0$,

$$\mathbf{U}_1 = \frac{\mathbf{V}_1 - (\mathbf{U}_0 \cdot \mathbf{V}_1)\mathbf{U}_0}{|\mathbf{V}_1 - (\mathbf{U}_0 \cdot \mathbf{V}_1)\mathbf{U}_0|}$$

The third vector is obtained from $\mathbf{V}_2$ by projecting out the $\mathbf{U}_0$ and the $\mathbf{U}_1$ components, thus producing a vector perpendicular to both of them,

$$\mathbf{U}_2 = \frac{\mathbf{V}_2 - (\mathbf{U}_0 \cdot \mathbf{V}_2)\mathbf{U}_0 - (\mathbf{U}_1 \cdot \mathbf{V}_2)\mathbf{U}_1}{|\mathbf{V}_2 - (\mathbf{U}_0 \cdot \mathbf{V}_2)\mathbf{U}_0 - (\mathbf{U}_1 \cdot \mathbf{V}_2)\mathbf{U}_1|}$$

Alternatively, you can compute $\mathbf{U}_2 = \mathbf{U}_0 \times \mathbf{U}_1$ and save cycles by not having to normalize as the other equation has you do.

Gram-Schmidt orthonormalization is equivalent to the QR decomposition for a matrix, where Q is orthogonal and R (the “right” matrix) is upper triangular. Let $M = [\mathbf{V}_0 \ \mathbf{V}_1 \ \mathbf{V}_2]$ be the matrix whose columns are the vectors we started with. Let $Q = [\mathbf{U}_0 \ \mathbf{U}_1 \ \mathbf{U}_2]$. By the construction, Q is necessarily orthogonal. The QR factorization is written as

$$\begin{aligned} [\mathbf{V}_0 \ \mathbf{V}_1 \ \mathbf{V}_2] &= [\mathbf{U}_0 \ \mathbf{U}_1 \ \mathbf{U}_2] \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ 0 & r_{11} & r_{12} \\ 0 & 0 & r_{22} \end{bmatrix} \\ &= [r_{00}\mathbf{U}_0 \quad r_{01}\mathbf{U}_0 + r_{11}\mathbf{U}_1 \quad r_{02}\mathbf{U}_0 + r_{12}\mathbf{U}_1 + r_{22}\mathbf{U}_2] \end{aligned}$$

From the actual construction of the orthonormal vectors, you can see that $r_{00} = |\mathbf{V}_0|$, $r_{01} = \mathbf{U}_0 \cdot \mathbf{V}_1$, $r_{11} = |\mathbf{V}_1 - (\mathbf{U}_0 \cdot \mathbf{V}_1)\mathbf{U}_0|$, $r_{02} = \mathbf{U}_0 \cdot \mathbf{V}_2$, $r_{12} = \mathbf{U}_1 \cdot \mathbf{V}_2$, and $r_{22} = |\mathbf{V}_2 - (\mathbf{U}_0 \cdot \mathbf{V}_2)\mathbf{U}_0 - (\mathbf{U}_1 \cdot \mathbf{V}_2)\mathbf{U}_1|$.

The fact that the input vectors are linearly independent guarantees that all of r_{00} , r_{11} , and r_{22} are positive. The QR factorization may be modified by factoring out the diagonal entries of R to produce $M = QDU$, where Q is orthogonal, D is a diagonal matrix with positive diagonal entries, and U is an upper-triangular matrix whose diagonal entries are all 1. The matrix U represents shearing, so QDU is a decomposition of M where shearing is applied first, scaling second, and rotation third.

17.5.2 EIGENDECOMPOSITION

If M is a symmetric matrix, then it is always possible to factor M into $M = RDR^T$, where R is an orthogonal matrix and D is a diagonal matrix. This is referred to as an *eigendecomposition* of M . The idea is that a nonzero vector $\mathbf{V}$ is an *eigenvector* of M with corresponding *eigenvalue* λ whenever $M\mathbf{V} = \lambda\mathbf{V}$. The eigenvector is special in that all that M does is change the length of the vector (with a direction flip when $\lambda < 0$). Symmetric matrices of size n are known to have n linearly independent eigenvectors $\mathbf{V}_i$ with corresponding eigenvalues λ_i (not necessarily distinct values). The equations $M\mathbf{V}_i = \lambda_i\mathbf{V}_i$, $0 \leq i < n$, are summarized in block form as

$$MR = M [\mathbf{V}_0 \quad \cdots \quad \mathbf{V}_{n-1}] = [\lambda_0\mathbf{V} \quad \cdots \quad \lambda_{n-1}\mathbf{V}_{n-1}] = RD$$

where R is the matrix whose columns are the eigenvectors and D is the diagonal matrix whose diagonal entries are the eigenvalues. The matrix M is then $M = RDR^T$.

Numerical methods for obtaining the eigendecomposition of a symmetric matrix are found in [PFTV88, GL93]. They are iterative in nature and, in fact, use the ideas of the QR algorithm for each iteration. For numerical robustness, the algorithms actually use a QL algorithm (where L is a lower-triangular matrix) as well as something called implicit shifting. The discussion in [GL93] is quite good regarding these matters.

17.5.3 POLAR DECOMPOSITION

The polar decomposition of a (square) matrix is $M = QS$, where Q is an orthogonal matrix and S is a symmetric matrix. Essentially, this is the best you can do when you want to factor M into “rotation” and “scale.” The matrix S represents the scaling, not in the standard coordinate system but rather in a rotated one. Since S is symmetric, it has an eigendecomposition $S = RDR^T$. Thus, $M = (QR)DR^T$. Assuming that you have decomposed the matrix so that Q and R are rotations, the decomposition says to rotate by R^T , scale in this new coordinate system by D , rotate back to the original system, and then rotate by Q .

This topic is covered in mathematical detail in the book [HJ85]. A practical discussion by Ken Shoemake is found in [Hec94, Section III.4].

17.5.4 SINGULAR VALUE DECOMPOSITION

The most general decomposition related to obtaining rotation and scaling information is the singular value decomposition (SVD). The (square) matrix is factored into $M = LSR$, where L and R are orthogonal matrices and S is a diagonal matrix whose diagonal entries are nonnegative. The diagonal values are referred to as the *singular values* of M .

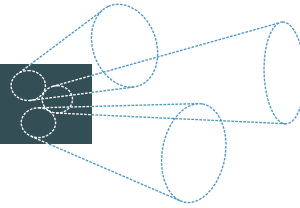
The books [HJ85] and [GL93] both have very good discussions on these topics, as does the article [Hec94, Section III.4]. The SVD uses an iterative approach, just as eigendecomposition and polar decomposition do. The SVD has a flavor all its own, but you can use eigendecompositions if you do not want to roll your own SVD. Specifically, notice that

$$MM^T = LSRR^T S^T L^T = LS^2 L^T$$

The right-hand side of this equation is an eigendecomposition of $M^T M$, where the columns of L are the eigenvectors for MM^T and the diagonal entries in S^2 are the corresponding eigenvalues. Similarly,

$$M^T M = R^T S^T L^T L S R = R^T S^2 R = Q S^2 Q^T$$

where $Q = R^T$. The columns of Q are the eigenvectors of $M^T M$ and the diagonal entries in S^2 are the corresponding eigenvalues.



OBJECT-ORIENTED INFRASTRUCTURE

A game engine is a large and complicated software system. The principles of object-oriented software engineering and large library design apply just as they would to any other large system. This chapter presents a review of some basic issues of object-oriented infrastructure. In addition, specific issues related directly to implementation of object-oriented support in the game engine are also addressed, including naming conventions and namespaces, run-time type information, single and multiple inheritance, templates (parameterized data types), shared objects and reference counting, streaming, and start-up and shutdown mechanisms. The final section is about a generic structure for an application layer in order to support the various components of a game engine with as much decoupling as possible.

18.1 OBJECT-ORIENTED SOFTWARE CONSTRUCTION

A good reference on object-oriented software engineering is [Mey88]. Extensive in-depth coverage of abstract data types including stacks, lists, strings, queues, maps, sets, trees, and graphs can be found in [Boo87]. Recent books have appeared addressing the issues of software development in the games industry. The one most directly related to the games industry is [RM03]. However, the principles of software engineering and object-oriented design apply equally as well to games as they do to any other area, so help yourself and browse some books on software engineering at your favorite library.

18.1.1 SOFTWARE QUALITY

The goal of software engineering is to help produce quality software, from the point of view both of the end users and of the software writers. The desired qualities in software fall into two categories:

1. External: Software is fast, reliable, and easy to use. The end users care about these qualities. End users also include team members who will use the code, so ease of use is important.
2. Internal: Software is readable, modular, and structured. The programmers care about these qualities.

The external qualities are the more important since the goal of software construction is building what a client wants. However, the internal qualities are key to attaining the external qualities. Object-oriented design is intended to deal with the internal, but the end result should satisfy the following external qualities:

- Correctness: the ability of software to exactly perform tasks, as defined by the requirements and specification.
- Robustness: the ability of software to function even in abnormal conditions.
- Extendability: the ease with which software may be adapted to changes of specifications.
- Reusability: the ability of software to be reused, in whole or in part, from new applications.
- Compatibility: the ease with which software products may be combined with others.
- Efficiency: the good use of hardware resources such as processor, memory, and storage, both in space and time.
- Portability: the ease with which software may be transferred to various hardware and software platforms.
- Verifiability: the ease of preparing test data and procedures for detecting and locating failures of the software.
- Integrity: the ability of software systems to protect their various components against unauthorized access and modification, whether or not the access or modification is intentional.
- Ease of use: the ease of learning how to use software, including executing the programs, preparing input data, interpreting output data, and recovering from exceptions.

Software maintenance is the process of modifying already existing code either to correct deficiencies, enhance efficiency, or extend the code to handle new or modi-

fied specifications. The following is a representative breakdown of maintenance time [Mey88]:

- Changes in user requirements (41.8%). Inevitable, but the large percentage is typically due to a lack of extendability.
- Changes in data formats (17.4%). Also inevitable since initial design may have lacked insight into how data might evolve.
- Emergency fixes (12.4%).
- Routine debugging (9.0%). For example, fixes need to be made, but the software can still run without them.
- Hardware changes (6.2%). Also inevitable, but isolation of hardware-dependent code can minimize these changes by encapsulation of the dependent code into device drivers.
- Documentation (5.5%). All of us are taught to do this as code is developed, but the reality is that the client always wants the code yesterday.
- Efficiency improvements (4.0%).

My experiences in development of systems and rewriting other companies' systems generally agree with this list. For games and game engine development, the changes in data format tend to happen infrequently, but when they do, you have a lot of changes to make (converters from one format to another).

18.1.2 MODULARITY

Modules are autonomous, coherent, robust, and organized packages. Not that this really defines what a module is, but all of us have an idea of what a module should be. The following five criteria should help in deciding what it means for a software construction method to be modular.

1. Decomposability. The design method helps decompose a problem into several subproblems whose solution may be pursued separately.
 - Example: Top-down design.
 - Counterexample: Initialization modules.
2. Composability. The design method supports production of software elements that may be freely combined to produce new systems.
 - Example: Math libraries.
 - Counterexample: Combined GUI and database libraries.
3. Understandability. The design method helps produce modules that can be separately understood by a human reader or can be understood together with a few other modules.

- Example: A math library with exported functions clearly specified and for which no other libraries are required for linking.
 - Counterexample: Sequentially dependent modules; module A depends on module B, module B depends on module C, and so on.
4. Continuity. A small change in the problem specification results in a change of just one (or a few) modules. Changes should not affect the architecture of the system.
 - Examples: Symbolic constants (do not hard-code numbers), the Principle of Uniform Reference (services of a module should be available through a uniform notation; in C++ this becomes a design question about public versus private members).
 - Counterexample: Failing to hide the data representation from the user when that representation may change later.
 5. Protection. The design method yields an architecture in which the effect of abnormal conditions at run time in a module remains confined to that module (or a few modules).
 - Example: Validation of input and output at their sources. This is the notion of preconditions and postconditions in abstract data types.
 - Counterexample: Undisciplined exceptions. An exception is a signal that is raised by one code block and handled in another, possibly remote part of the system. This separates algorithms for normal cases from error processing in abnormal cases, but the mechanism violates the criterion of confining the abnormal conditions to the module. This also violates the continuity criterion.

The five criteria lead to five principles that should be followed to ensure modularity. The criteria that lead to each principle are listed in parentheses.

1. Linguistic modular units. Modules must correspond to syntactic units in the language used (decomposability, composability, protection).
2. Few interfaces. Every module should communicate with as few others as possible (continuity, protection).
3. Small interfaces. If two modules must communicate, they should exchange as little information as possible. This is termed *weak coupling* (continuity, protection).
4. Explicit interfaces. Whenever two modules communicate, this must be obvious from the text of the modules. This is termed *direct coupling* (decomposability, composability, continuity, understandability).
5. Information hiding. All information about the module should be private unless it is declared public (continuity, not necessarily protection).

The principle of information hiding is sometimes debated as an evil that is not necessary. I have heard this from a few commercial game developers. The issue at hand is that during development certain information might be hidden from you that,

if visible, would make debugging a lot easier and faster. The presumed solution is to make everything public to the developers. I strongly disagree with this stance. Many view each game as throwaway code. With the increasing complexity of games and the enormous amount of source code that goes into them, companies need to reuse code and to ensure that the code, when originally written, is of good quality. If debugging a module is difficult, I would question the quality of the module and require that it be improved rather than expose every variable possible.

The Open-Closed Principle

This is one final requirement for a good modular decomposition. It states that a module must be both *open* and *closed*.

- Open module: The module is still available for extension. For example, it is still possible to add fields to data structures or to add new functions that operate on the structures.
- Closed module: The module is available for use by other modules. This assumes that the module has a well-defined, stable interface, with the emphasis being on “stable.” For example, such a module would be compiled into a library.

At first glance, being both open and closed appears to be contradictory. If the public interface to a module remains constant, but the internal implementations are changed, the module may be considered open and closed (it has been modified, but dependent code does not need to be changed or recompiled). However, most modifications of modules are to add new functionality. The concept of *inheritance* allows for open-closed modules. A base class is closed in itself, but a derived class can add new members and functionality. In this sense the base class plus derived class represent the openness.

18.1.3 REUSABILITY

Reusability is a basic issue in software engineering. Why spend time designing and coding an algorithm when it probably already exists elsewhere? But this question does not have a simple answer. It is easy to find already written code for searching and sorting lists, handling stacks, and other basic data structure manipulations (STL is now the prime example). However, other factors may compound the issue. Some companies provide libraries that have capabilities you need, but to use the libraries you need to purchase a license and possibly pay royalties. If the acquired components have bugs in them, you must rely on the provider to fix them, and that will probably not occur in the time frame in which you need the repairs.

At least in your local environment, you can attempt to maximize reuse of your own components. Here are some issues for module structures that must be resolved to yield reusable components:

- Variation in types. The module should be applicable to structures of different types. Templates or parameterized data types can help here.
- Variation in data structures and algorithms. The actions performed during an algorithm might depend on the underlying structure of the data. The module should allow for handling variations of the underlying structures. Overloading can help here.
- Related routines. The module must have access to routines for manipulating the underlying data structure.
- Representation independence. The module should allow a user to specify an operation without knowing how it is implemented or what underlying data structures have been used. For example,

```
x_is_in_table_t = search(x,t);
```

is a call to search for item *x* in a table *t* and return the (Boolean) result. If many types of tables are to be searched (lists, trees, files, etc.), it is desirable not to have massive control structures such as

```
if ( t is of type A )
    apply search algorithm A
else if ( t is of type B )
    apply search algorithm B
else if ...
```

whether it be in the module code or in the client code. Overloading and polymorphism can help here.

- Commonality within subgroups. Extract commonality, extract commonality, extract commonality! Avoid the repetition of similar blocks of code because if a change is required in one block, it is probably also required in the other similar blocks, which will require a lot of time spent on maintenance. Build an abstract interface that doesn't expose the underlying data structures.

18.1.4 FUNCTIONS AND DATA

Which comes first, functions or data? The key element in answering this question is the problem of extendability and, in particular, the principle of continuity. During the full life cycle, functions tend to change quite a bit since requirements on the system also tend to change regularly. However, the data on which the functions operate tends

to be persistent and change very little. The object-oriented approach is to concentrate on building modules based on objects.

A classical design method is the top-down functional approach—specifying the system’s abstract function, then applying stepwise refinement to smaller, more manageable functions. The approach is logical and well organized, and encourages orderly development. The drawbacks are as follows:

- The method ignores the evolutionary nature of software systems. The problem is continuity. The top-down approach yields short-term convenience, but as the system changes, there will be constant redesigning, with a large potential for long-term disaster.
- The notion of a system being characterized by one function is questionable. An operating system is the classic case of a system not characterized by a single “main” function. *Real systems have no top.*
- The method does not promote reusability. The designers tend to decompose the functions based on current specifications. The subroutines are reflections of the initial design. As the system evolves, the subroutines may no longer be relevant to the new requirements.

18.1.5 OBJECT ORIENTATION

Object-oriented design leads to software architectures based on the objects every system or subsystem manipulates rather than “the function” it is meant to ensure. Issues are

- How to find the objects. A well-organized software system may be viewed as an *operational model* of some aspect of the world. The software objects will simply reflect the real-world objects.
- How to describe the objects. The standard approach to describing objects is through *abstract data types*. Specification for an abstract data type involves *types* (type becomes a parameter of the abstraction), *functions* (what operations are applied), *preconditions* (these must be satisfied before operations are applied), *postconditions* (these must be satisfied after operations are applied), and *axioms* (how compositions of the functions behave).

Object-oriented design is also the construction of software systems as structured collections of abstract data type implementations. Issues are

- Object-based modular structure. Systems are modularized on the basis of their data structures.
- Data abstraction. Objects should be described as implementations of abstract data types.

- Automatic memory management. Unused objects should be deallocated by the underlying language system, without programmer intervention.
- Classes. Every nonsimple type is a module, and every high-level module is a type. This is implemented as the *one-class-per-module* paradigm.
- Inheritance. A class may be defined as an extension or restriction of another.
- Polymorphism and dynamic binding. Program entities should be permitted to refer to objects of more than one class, and operations should be permitted to have different realizations in different classes.
- Multiple and repeated inheritance. It should be possible to declare a class as heir to more than one class, and more than once to the same class.

Whether or not a language can support all the various features mentioned in this section is questionable. Certainly, SmallTalk and Ada make claims that they are fully featured. However, fully featured languages come at a price in performance. The object-oriented code that accompanies this book is written in C++. While not a “pure” object-oriented language, C++ supports the paradigm fairly well, yet allows flexibility in dealing with situations where performance is important. One of the common fallacies about C++ is that its performance is unacceptable compared to that of C. Keep in mind that a compiler is a large software system itself and is susceptible, just as any other large system, to being poorly implemented. Current-generation C++ compilers produce code that is quite compact and fast. For a reference book on C++, see [Str00]. For an extensive set of examples illustrating the features and use of C++, see [LLM05].

18.2 STYLE, NAMING CONVENTIONS, AND NAMESPACES

One of the software engineering goals mentioned previously is that code should be readable. In an environment with many programmers developing small pieces of a system, each programmer tends to have his or her own style, including choice of identifier names, use of white space, alignment and indentation of code, placement of matching braces, and internal comments. If a team of programmers develops code that will be read both internally (by other team members) and externally (by paying clients), ideally the code should have as consistent a style as possible purely from the point of view of readability. Inconsistent style distracts from the client’s main purpose—to understand and use the code for his or her own applications. A management-imposed style certainly is a possibility, but beware of the potential religious wars. Many of today’s C++ programmers learned C first and learned their programming style at that time. Although a lot of the conventions in that language are not consistent with an object-oriented philosophy, the programmers are set in their ways and will still use what they originally learned.

Naming conventions are particularly important so that a reader of the code knows what to expect across multiple files that were written by multiple programmers. One of the most useful naming conventions used in the code on the CD-ROM that accompanies this book allows the reader to distinguish between class members, local variables, and global variables, including whether they are nonstatic or static. This makes it easy to determine where to look for definitions of variables and to understand their scope.¹ Moreover, the identifier names have type information encoded in them. The embedded information is not as verbose as Microsoft's Hungarian notation, but it is sufficient for purposes of readability and understandability of the code.

Because a game engine, like any other large library, will most likely be integrated with software libraries produced by other teams, whether internal or external, there is the possibility of clashes of class names and other global symbols. Chances are that you have named your matrix class `Matrix` and so has someone else who has produced header files and libraries for your use. Normally, someone has to make a name change to avoid the clash. C++ provides the concept of namespace to support avoiding the clashes, but a method that is popular among many library producers is to use a prefix on class names and global symbols in hopes that the prefix is unique among all packages that will be integrated into the final product. The namespace construct implicitly mangles the class names, whereas the manual selection of a prefix makes the mangling explicit.

The conventions used for the accompanying code are as follows. In the first edition of the book, which shipped with Wild Magic Version 0.1, I used the explicit scoping by the `Mgc` prefix. Since then I have abandoned that verbosity and switched to using namespaces. This book ships with Wild Magic 4.0, in which I use the namespace `Wm4`. Version 3 of Wild Magic used `Wm3`. The different namespaces actually had a pleasant, practical side effect. I had to convert data files from the formats used in Wild Magic 3 to the formats used in Wild Magic 4. Having distinct namespaces allowed me to build reformatting applications that linked in both versions of the library. Although you might think this obvious, when working on NetImmerse many years ago, we had the problem of data file conversion every time we shipped a new version, but we used explicit scoping (with the prefix `Ni`) in both the old and new versions. Format conversion was generally a nightmare.

Regarding identifiers in the source code, function names are capitalized; if multiple words make up the name, each distinct word is capitalized. For example, given a class that represents a string, a class member function to access the length of the string would be named `GetLength`. Identifier names are capitalized in the same way that function names are, but with prefixes. Nonstatic class data members are prefixed with `m_`, and static class data members are prefixed with `ms_`. The *m* refers to “member” and the *s* indicates “static.” A static local variable is prefixed with `s_`. A global

1. The evolution of source code browsers and databases in development tools makes finding the definitions of variables quite easy. However, not everyone will have such tools, so I doubt if I will change my naming conventions anytime soon!

Table 18.1 Encoding for the various types to be used in identifier names.

<i>Type</i>	<i>Encoding</i>	<i>Type</i>	<i>Encoding</i>
char	c	unsigned char	uc
short	s	unsigned short	us
int	i	unsigned int	ui
long	l	unsigned long	ul
float	f	double	d
pointer	p	smart pointer	sp
reference	r	array	a
enumerated type	e	class variable	k
template	t	function pointer	o
void	v	handle	h
interface (DX9)	q		

variable is prefixed with `g_`, and a static global variable is prefixed with `gs_`. The type of the variable is encoded and is a prefix to the identifier name, but follows the underscore (if any) for member or global variables. Table 18.1 lists the various encoding rules.

To avoid excessively long names, loop counters are also allowed to use *j*, *k*, and other one-letter names. For Wild Magic 4.1, I plan on replacing loop counters with type `ctri_t` for a signed integer counter, and with type `ctru_t` for an unsigned integer count. These will be defined differently depending on the natural word size for a platform (32-bit systems or 64-bit systems).

Also, I now allow some data members to be in public scope if there are no side effects needed when in the “set” or “get” semantics. These members have variable names that follow the rules for function identifiers. If the data member is a constant or a static constant, the name has all letters capitalized.

Identifier names do not use underscores, except for the prefixes as described earlier. Class constants are capitalized and may include underscores for readability. Combinations of the encodings are also allowed; for example,

```
unsigned int* auArray = new unsigned int[16];
void ReallocArray (int iQuantity, unsigned int*& rauArray)
{
    delete[] rauArray;
    rauArray = new unsigned int[iQuantity];
}
```

```

short sValue;
short& rsValue = sValue;
short* psValue = &sValue;

class SomeClass
{
public:
    SomeClass ();
    SomeClass (const SomeClass& rkObject);

protected:
    enum { NOTHING, SOMETHING, SOMETHING_ELSE };
    int m_eSomeFlag;
};

```

The rules of style in the code are not listed here and can be inferred from reading any of the source files.

18.3 RUN-TIME TYPE INFORMATION

Polymorphism provides abstraction of functionality. A polymorphic function call can be made regardless of the true type of the calling object. But there are times when you need to know the type of the polymorphic object, or you need to determine if the object's type is derived from a specified type—for example, to safely typecast a base class pointer to a derived-class pointer, a process called *dynamic typecasting*. *Run-time type information* (RTTI) provides a way to determine this information while the program is executing.

18.3.1 SINGLE-INHERITANCE SYSTEMS

A single-inheritance, object-oriented system consists of a collection of directed trees where the vertices represent classes and the edges represent inheritance. Suppose vertex V_0 represents class C_0 and vertex V_1 represents class C_1 . If C_1 inherits from C_0 , then the directed edge from V_1 to V_0 represents the inheritance relationship between C_1 and C_0 . The directed edges indicate an *is-a* relationship. Figure 18.1 shows a simple single-inheritance hierarchy.

The root of the tree is POLYGON. RECTANGLE is a POLYGON, and SQUARE is a RECTANGLE. Moreover, SQUARE is a POLYGON indirectly. TRIANGLE is a POLYGON; EQUILATERALTRIANGLE is a TRIANGLE; and RIGHTTRIANGLE is a TRIANGLE. However, SQUARE is not a TRIANGLE, and RIGHTTRIANGLE is not an EQUILATERALTRIANGLE.

An RTTI system is a realization of the directed trees. The basic RTTI data type stores any class-specific information that an application might require at run time.

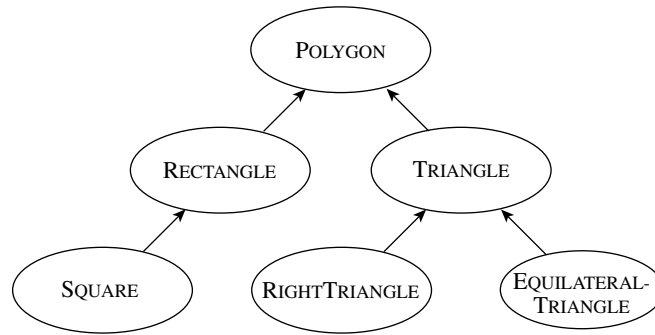


Figure 18.1 Single-inheritance hierarchy.

It also stores a link to the base class (if any) to allow an application to determine if a class is inherited from another class. The simplest representation stores no class information and only the link to the base class. However, it is useful to store a string encoding the name of the class. In particular, the string will be used in the streaming system that is described later. The string may also be useful for debugging purposes in quickly identifying the class type.

```
class Rtti
{
public:
    Rtti (const char* acName, const Rtti* pkBaseType)
    {
        m_acName = acName;
        m_pkBaseType = pkBaseType;
    }

    const char* GetName () const
    {
        return m_acName;
    }

    bool IsExactly (const Rtti& rkType) const
    {
        return &rkType == this;
    }

    bool IsDerived (const Rtti& rkType) const
```

```

{
    const Rtti* pkSearch = this;
    while (pkSearch)
    {
        if (pkSearch == &rkType)
        {
            return true;
        }
        pkSearch = pkSearch->m_pkBaseType;
    }
    return false;
}

private:
    const char* m_acName;
    const Rtti* m_pkBaseType;
};

```

The root class `Object` in an inheritance tree must contain basic support for the RTTI system. Minimally, the class is structured as

```

class Object
{
public:
    static const Rtti TYPE;

    virtual const Rtti& GetType () const
    {
        return TYPE;
    }

    bool IsExactly (const Rtti& rkType) const
    {
        return GetType().IsExactly(rkType);
    }

    bool IsDerived (const Rtti& rkType) const
    {
        return GetType().IsDerived(rkType);
    }

    bool IsExactlyTypeOf (const Object* pkObj) const
    {
        return pkObj && GetType().IsExactly(pkObj->GetType());
    }
}

```



```

bool IsDerivedTypeOf (const Object* pObj) const
{
    return pObj && GetType().IsDerived(pObj->GetType());
}
};

```

In addition to the RTTI support, my implementation of an object system has template functions for static typecasting and dynamic typecasting. These are

```

template <class T>
T* StaticCast (Object* pObj)
{
    return (T*)pObj;
}

template <class T>
const T* StaticCast (const Object* pObj)
{
    return (const T*)pObj;
}

template <class T>
T* DynamicCast (Object* pObj)
{
    return pObj && pObj->IsDerived(T::TYPE) ? (T*)pObj : 0;
}

template <class T>
const T* DynamicCast (const Object* pObj)
{
    return pObj && pObj->IsDerived(T::TYPE) ? (const T*)pObj : 0;
}

```

Each derived class in the inheritance tree has a static type object TYPE and must minimally be structured as

```

class DerivedClass : public BaseClass
{
public:
    static const Rtti TYPE;
    virtual const Rtti& GetType () const { return TYPE; }
};

```

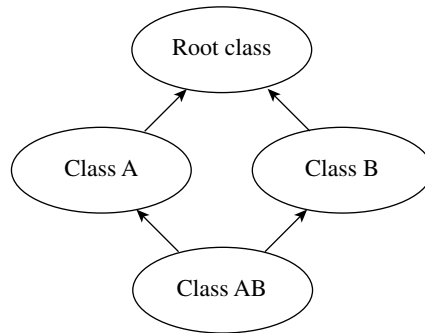


Figure 18.2 Multiple-inheritance hierarchy. Class AB inherits from both class A and class B and indirectly inherits from the root class.

where `BaseClass` is, or is derived from, `Object`. Note that the unique identification is possible since the static `TYPE` members all have distinct addresses in memory at run time. The source file for the derived class must contain

```
const Rtti DerivedClass::TYPE("Wm4.DerivedClass",&BaseClass::TYPE);
```

In my implementation, I include the namespace as a prefix to the class name. This allows the RTTI system to work with applications built from multiple libraries, each using its own namespace but relying on the RTTI system in mine.

18.3.2 MULTIPLE-INHERITANCE SYSTEMS

A multiple-inheritance, object-oriented system consists of a collection of directed acyclic graphs where the vertices represent classes and the edges represent inheritance. Suppose vertices V_i represent classes C_i for $i = 0, 1, 2$. If C_2 inherits from both C_0 and C_1 , then V_2 has directed edges to both V_0 and V_1 that represent the multiple inheritance. Figure 18.2 shows a multiple-inheritance hierarchy.

An RTTI system in the context of multiple inheritance is a realization of the directed acyclic graphs. While the RTTI data type for a singly inherited system has a single link to a base class, the RTTI data type for a multiply inherited system requires a list of links to the base classes (if any). The simplest representation stores no class information and only the links to the base classes. To support a to-be-determined number of base classes, the C-style ellipses are used in the constructor, thus requiring standard argument support. For most compilers, including `cstdarg` gives access to the macros for parameter parsing.

```

class Rtti
{
public:
    Rtti (const char* acName, int iNumBaseClasses,...)
    {
        m_acName = acName;
        if (iNumBaseClasses == 0)
        {
            m_iNumBaseClasses = 0;
            m_apkBaseType = 0;
        }
        else
        {
            m_iNumBaseClasses = iNumBaseClasses;
            m_apkBaseType = new Rtti*[iNumBaseClasses];
            va_list list;
            va_start(list,iNumBaseClasses);
            for (int i = 0; i < iNumBaseClasses; i++)
            {
                m_apkBaseType[i] = va_arg(list, const Rtti*);
            }
            va_end(list);
        }
    }

    ~Rtti ()
    {
        delete[] m_apkBaseType;
    }

    bool IsDerived (const Rtti& rkType) const
    {
        if (&rkType == this)
        {
            return true;
        }
        for (int i = 0; i < m_iNumBaseClasses; i++)
        {
            if (m_apkBaseType[i]->IsDerived(rkType))
            {
                return true;
            }
        }
        return false;
    }
}

```

```

// ... other public functions ...

private:
    const char* m_acName;
    unsigned int m_uiNumBaseClasses;
    Rtti** m_apkBaseType;
};

```

The root class in a single-inheritance tree provides the member functions for searching the directed tree to determine if one class is the same or derived from another class. A technical problem with a multiple-inheritance directed graph is that there may be more than one vertex with no edges; that is, the hierarchy may have multiple root classes. To avoid this situation, always provide a single root class whose sole job is to provide an interface for any systems used by the entire inheritance graph. The root class in the multiple-inheritance graph is structured exactly as in the single-inheritance tree.

The derived classes still provide the same static RTTI member and a virtual function to access its address. For example, consider

```

class DerivedClass : public BaseClass0, public BaseClass1
{
public:
    static const Rtti TYPE;
    virtual const Rtti& GetType () const { return TYPE; }
};

```

where both BaseClass0 and BaseClass1 are either Object or derived from Object. The source file for this derived class must contain

```

const Rtti Derived::TYPE("Wm4.DerivedClass",2,
    &BaseClass0::TYPE,&BaseClass1::TYPE);

```

18.3.3 MACRO SUPPORT

Macros can be used to simplify use by an application and to hide the verbosity of the code. I do not create a lot of macros—only enough to get the job done for my object system. These are found in the graphics library, the ObjectSystem folder, files Wm4Main.mcr, Wm4NameID.mcr, Wm4Rtti.mcr, and Wm4Stream.mcr. The Main macros are for premain initialization and postmain termination of objects. The NameID macros are for providing unique names and identifiers for objects in the system. The Rtti macros are for the run-time type information system. The Stream macros are for the streaming system for loading objects from files on disk and saving objects to files on disk.

18.4 TEMPLATES

Templates, sometimes called *parameterized data types*, are used to share code among classes that all require the same structure. The classic example is a stack of objects. The operations for a bounded stack are Push, Pop, IsEmpty, IsFull, and GetTop (read top element without popping the stack). The operations are independent of the type of object stored on the stack. A stack could be implemented for both int and float, each using array storage for the stack elements. The only difference between the two implementations is that the integer stack code uses an array of int and the float stack code uses an array of float. A template can be used instead so that the compiler generates object code for each type requested by an application.

```
template <class T> class Stack
{
public:
    Stack (int iStackSize)
    {
        m_iStackSize = iStackSize;
        m_iTop = -1;
        m_atStack = new T[iStackSize];
    }

    ~Stack () { delete[] m_atStack; }

    bool Push (const T& rkElement)
    {
        if (m_iTop < m_iStackSize)
        {
            m_atStack[++m_iTop] = rkElement;
            return true;
        }
        return false;
    }

    bool Pop (T& rkElement)
    {
        if (m_iTop >= 0)
        {
            rkElement = m_atStack[m_iTop--];
            return true;
        }
        return false;
    }
}
```

```

bool GetTop (T& rkElement) const
{
    if (m_iTop >= 0)
    {
        rkElement = m_atStack[m_iTop];
        return true;
    }
    return false;
}

bool IsEmpty () const { return m_iTop == -1; }
bool IsFull () const { return m_iTop == m_iStackSize-1; }

protected:
    int m_iStackSize;
    int m_iTop;
    T* m_atStack;
};

```

Macros could be used to generate code for different types, but the macros are not typesafe and are susceptible to side effects. Although it is possible to implement the stack code for both `int` and `float`, this poses a problem for code maintenance. If one file changes, the other must be changed accordingly. The maintenance issue is magnified even more so when there is a large number of types sharing the same code. Templates provide a way of localizing those changes to a single file.

Templates are a good choice for container classes for various data structures such as stacks, arrays, lists, and so on. The Standard Template Library is now a part of C++ and may be used by the game engine. One problem to be aware of when dealing with a container of objects (in this case, objects of type `Object`) is that certain side effects of the class are necessary, especially in construction and destruction. If a standard template library container class has a need to resize itself, it might do so by creating an array of the new size, placing a memory copy of the old array into the new array, and then deleting the old array. This scheme has the implicit assumption that the underlying data is native. If the data consists of class objects where the constructor allocates memory and the destructor deallocates memory, the memory copy causes memory leaks and misses side effects that occur because of object construction or destruction. This will definitely be the case for shared objects and reference counting, the topic of the next section. If the Standard Template Library does not support side effects, the game engine code will need to implement its own template container classes. The STL mainly supports side effects, but for even more complicated interactions between objects and the template library, see the ever popular Boost library [Lib].

18.5 SHARED OBJECTS AND REFERENCE COUNTING

Sharing of objects is natural in a game engine. Models that contain a lot of data might be shared to minimize memory use. Renderer state can also be shared, particularly when textures are shared among objects. It is unlikely that a game engine can be implemented in a way to *manually* manage shared objects without losing some along the way (object leaking) or destroying some while still in use by other objects (premature destruction). Therefore, a more automated system is required to assist in the bookkeeping of sharing. A popular system is to add a reference counter to the root class object. Each time an object is shared (referenced) by another object, the reference counter is incremented. Each time an object is finished sharing with another, the reference counter is decremented. Once the reference counter decreases to zero, the object is no longer referenced within the system, and it is deleted.

The details of reference counting can be exposed so that the application is responsible for adjusting the reference counter, but this mechanism places great faith in the programmer to properly manage the objects. Another possibility is to implement a *smart pointer* system that adjusts the reference counter internally while still allowing the application to intervene in cases that require special handling. Thus, the burden of proper management of shared objects is mostly taken from the programmer. The Java programming language incorporates this system. The Boost libraries [Lib] also provide support for smart pointers of various flavors.

In addition to run-time type information, the root class `Object` now includes the following code to support reference counting:

```
public:
    void IncrementReferences ()
    {
        m_iReferences++;
    }

    void DecrementReferences ()
    {
        if (--m_iReferences == 0)
        {
            delete this;
        }
    }

    int GetReferences () const
    {
        return m_iReferences;
    }
```

```

static stdext::hash_map<unsigned int, Object*>* InUse;
static void PrintInUse (const char* acFilename,
    const char* acMessage);
private:
    int m_iReferences;

```

The static hash map keeps track of objects currently in the system. The hash map is initially empty at program execution time. At program termination, if the hash map is not empty, the contents may be printed to a file using the `PrintInUse` function. This is useful for tracking object leaks and for letting the application writer know that he did not release all the objects he should have.

The smart pointer system is built on top of this system and uses templates:

```

template <class T> class Pointer
{
public:
    // construction and destruction
    Pointer (T* pkObject = 0)
    {
        m_pkObject = pkObject;
        if (m_pkObject)
        {
            m_pkObject->IncrementReferences();
        }
    }

    Pointer (const Pointer& rkPointer)
    {
        m_pkObject = rkPointer.m_pkObject;
        if (m_pkObject)
        {
            m_pkObject->IncrementReferences();
        }
    }

    ~Pointer ()
    {
        if (m_pkObject)
        {
            m_pkObject->DecrementReferences();
        }
    }

    // implicit conversions

```



```

operator T* () const { return m_pkObject; }
T& operator* () const { return *m_pkObject; }
T* operator-> () const { return m_pkObject; }

// assignment
Pointer& operator= (T* pkObject)
{
    if (m_pkObject != pkObject)
    {
        if (pkObject)
        {
            pkObject->IncrementReferences();
        }

        if (m_pkObject)
        {
            m_pkObject->DecrementReferences();
        }

        m_pkObject = pkObject;
    }
    return *this;
}

Pointer& operator= (const Pointer& rkReference)
{
    if (m_pkObject != rkPointer.m_pkObject)
    {
        if (rkPointer.m_pkObject)
        {
            rkPointer.m_pkObject->IncrementReferences();
        }

        if (m_pkObject)
        {
            m_pkObject->DecrementReferences();
        }

        m_pkObject = rkPointer.m_pkObject;
    }
    return *this;
}

// comparisons

```

```

bool operator== (T* pkObject) const
{
    return m_pkObject == pkObject;
}

bool operator!= (T* pkObject) const
{
    return m_pkObject != pkObject;
}

bool operator== (const Pointer& rkReference) const
{
    return m_pkObject == rkPointer.m_pkObject;
}

bool operator!= (const Pointer& rkReference) const
{
    return m_pkObject != rkPointer.m_pkObject;
}

protected:
    // the shared object
    T* m_pkObject;
};

```

The assignment operator must compare the pointer values first before adjusting reference counting to guard against assignments of the type

```

Pointer<Object> spkPointer = new Object;
spkPointer = spkPointer;

```

Although you might not have such lines of code in your program, indirectly you may have a situation where an assignment occurs from an object to itself.

The constructor for `Object` sets the references to zero. The constructor for `Pointer` increments the references to one. If the initial comparison were not present in the assignment operator, the call to `DecrementReferences` would decrement the references to zero, then destroy the object. Consequently, the pointer `rkPointer.m_pkObject` points to a memory block no longer owned by the application, and the pointer `m_pkObject`, the target of the assignment, will point to the same invalid block. The call to `IncrementReferences` will write to the invalid block—an error. Although such a statement is unlikely in a program, the situation might arise unexpectedly due to pointer aliasing.

The `Object` base class defines a smart pointer type, and each derived class does the same.

```
typedef Pointer<Object> ObjectPtr;
typedef Pointer<DerivedClass> DerivedClassPtr;
```

This is for convenience in programming.

There might be a need to typecast a smart pointer to a pointer or smart pointer. For example, class *Node*, the internal node representation for scene graphs, is derived from *Spatial*, the leaf node representation for scene graphs. Polymorphism allows the assignment

```
Node* pkNode = <some node in scene graph>;
Spatial* pkObject = pkNode;
```

Abstractly, a smart pointer of type *NodePtr* is derived from *SpatialPtr*, but the language does not support such mirroring of class derivation trees. The use of implicit operator conversions in the smart pointer class guarantees a side effect that makes it appear as if the derivation really does occur. For example,

```
// This code is valid.
NodePtr spkNode = <some node in scene graph>;
SpatialPtr spkObject = spkNode;

// This code is not valid when class A is not derived from
// class B.
APtr spkAObject = new A;
BPtr spkBObject = spkAObject;
```

The implicit conversions also support comparison of smart pointers to null, just like regular pointers:

```
NodePtr spkNode = <some node in scene graph>;
SpatialPtr spkChild = spkNode->GetChildAt(2);
if (spkChild)
{
    <do something with spkChild>;
}
```

A simple example illustrating the use and cleanup of smart pointers follows. The class *Node* stores an array of smart pointers for its children.

```
NodePtr spkNode = <some node in scene graph>;
Node* pkNode = new Node;           // pkNode references = 0
NodePtr spkChild = pkNode;         // pkNode references = 1
spkNode->AttachChild(spkChild);     // pkNode references = 2
```

```

spkNode->DetachChild(spkChild); // pkNode references = 1
spkChild = 0;                  // pkNode references = 0,
                                // destroy it

```

This illustrates how to properly terminate use of a smart pointer. In this code the call `delete spkChild` would work just fine. However, if the object that `spkChild` points to has a positive reference count, explicitly calling the destructor forces the deletion, and the other objects that were pointing to the same object now have dangling pointers. If instead the smart pointer is assigned 0, the reference count is decremented and the object pointed to is not destroyed if there are other objects referencing it. Thus, code like the following is safe:

```

NodePtr spkNode = <some node in scene graph>;
Node* pkNode = new Node;           // pkNode references = 0
NodePtr spkChild = pkNode;         // pkNode references = 1
spkNode->AttachChild(spkChild);     // pkNode references = 2
spkChild = 0;                      // pkNode references = 1,
                                    // no destruction

```

Also note that if the assignment of 0 to the smart pointer is omitted in this code, the destructor for the smart pointer is called and the reference count for `pkNode` still is decremented to one.

Some other guidelines that must be adhered to when using smart pointers follow. Smart pointers apply only to dynamically allocated objects, not to objects on the stack. For example,

```

void MyFunction ()
{
    Node kNode;                    // kNode references = 0
    NodePtr spkNode = &kNode;      // kNode references = 1
    spkNode = 0;                   // kNode references = 0,
                                    // kNode is deleted
}

```

is doomed to failure. Since `kNode` is on the stack, the deletion implied in the last statement will attempt to deallocate stack memory, which is an error.

Using smart pointers as function parameters or returning them as the result of a function call also has its pitfalls. The following example illustrates the dangers:

```

void MyFunction (NodePtr spkNode)
{
    <do nothing>;
}

```

```
Node* pNode = new Node;
MyFunction(pNode);
// pNode now points to invalid memory
```

On allocation `pNode` has zero references. The call to `MyFunction` creates an instance of a `NodePtr` on the stack via the copy constructor for that class. That call increments the reference count of `pNode` to one. On return from the function, the instance of `NodePtr` is destroyed and, in the process, `pNode` has zero references and it too is destroyed. However, the following code is safe:

```
Node* pNode = new Node;    // pNode references = 0
NodePtr spkNode = pNode;   // pNode references = 1;
MyFunction(spkNode);       // pNode references increase to 2,
                           // then decrease to 1
// pNode references = 1 at this point
```

A related problem is

```
NodePtr MyFunction ()
{
    Node* pReturnNode = new Node; // references = 0;
    return pReturnNode;
}

Node* pNode = MyFunction();
// pNode now points to invalid memory
```

A temporary instance of a `NodePtr` is implicitly generated by the compiler for the return value of the function. The copy constructor is called to generate that instance, so the reference count of `pNode` is one. The temporary instance is no longer needed and is implicitly destroyed and, in the process, `pNode` has zero references and it too is destroyed. The following code is safe:

```
NodePtr spkNode = MyFunction();
// spkNode.m_pObject has one reference
```

The temporary instance increases the reference count of `pReturnNode` to one. The copy constructor is used to create `spkNode`, so the reference count increases to two. The temporary instance is destroyed, and the reference count decreases to one.

18.6 STREAMING

Persistence of storage is a requirement for a game engine. Game content is typically generated by a modeling tool and must be exported to a format that the game applica-

tion can import. The game application itself might have a need to save its data so that it can be reloaded at a later time. *Streaming* of data refers to the process of mapping data between two media, typically disk storage and memory. In this section, we will discuss transfers between disk and memory, but the ideas directly apply to transfers between memory blocks (which supports transfers across a network).

A scene graph is considered to be an abstract directed graph of objects (of base type `Object`). The nodes of the graph are the objects, and the arcs of the graph are pointers between objects. Each object has nonobject members, in particular any members of a native data type (integer, float, string, etc.). The abstract graph must be saved to disk so that it can be re-created later. This means that both the graph nodes and graph arcs must be saved in some reasonable form. Moreover, each graph node should be saved exactly once. The process of saving a scene graph to disk is therefore equivalent to creating a list of the unique objects in the graph, saving them to disk, and in the process saving any connections between them. If the graph has multiple connected components, then each component must be traversed and saved. Support for saving multiple abstract objects is easy to implement. The class `Stream` provides the ability to assemble a list of *top-level* objects to save. Typically, these are the roots of scene graphs, but they can be other objects whose state needs to be saved. To support loading the file and obtaining the same list of top-level objects, an identifying piece of information must be written to disk before each abstract graph corresponding to a top-level object. A simple choice is to write a string to disk.

18.6.1 THE STREAM API

The class that exists to manage the streaming process is `Stream`. The relevant public portion of the interface is

```
class Stream
{
public:
    // construction and destruction
    Stream ();
    ~Stream ();

    // the objects to process, each object representing an entry
    // into a connected component of the abstract graph
    bool Insert (Object* pObj);
    bool Remove (Object* pObj);
    void RemoveAll ();
    int GetObjectCount ();
    Object* GetObjectAt (int i) const;
    bool IsTopLevel (Object* pObj);
```

```

// Memory loads and saves. Stream does not assume
// responsibility for the char arrays. The application must
// manage the input acBuffer for the call to Load and delete
// the output racBuffer for the call to Save.
bool Load (char* acBuffer, int iSize);
bool Save (char*& racBuffer, int& riSize);

// File loads and saves.
bool Load (const char* acFilename);
bool Save (const char* acFilename);

// Support for disk usage.
int GetDiskUsed () const;
};

```

A Stream object manages a list of top-level objects. Objects are inserted into the list by `Insert` and removed from the list by `Remove` or `RemoveAll`. The function `GetObjectCount` returns the number of objects in the top-level list. The function `GetObjectAt(int)` returns the *i*th object in the list. The function `IsTopLevel` is mainly used internally by Stream, but may be called by an application as a check for existence of an object in the top-level list.

Streaming to and from disk is supported by the load/save functions that take a file name (character string) as input. The other load/save functions are for streaming to and from a memory block. The return value is true if and only if the function call was successful.

The function call `GetDiskUsed` computes how much disk space the top-level objects will use, not counting the file header that is used in the Wild Magic scene file format. This function is also used internally by the file `Save` function to allocate a memory block of the appropriate size, stream the top-level objects to that block, and then write the block with a single call to a low-level file writing function. The intent is to avoid expensive disk operations that might occur if writes are made on a member-by-member basis for each object. Every class derived from `Object` must implement `GetDiskUsed`.

The typical usage for disk streaming is shown in the next code block:

```

// Save a list of objects.
Stream kOutputStream;
kOutputStream.Insert(pkObject1);
:
kOutputStream.Insert(pkObjectN);
kOutputStream.Save("myfile.wmof");

// Load a list of objects.
Stream kInStream;
bool bLoaded = kInStream.Load("myfile.wmof");

```

```

if (bLoaded)
{
    for (int i = 0; i < kInStream.GetObjectCount(); i++)
    {
        ObjectPtr spkObject = kInStream.GetObjectAt(i);
        // Use prior knowledge of the file contents and statically
        // cast the objects for further use by the application.
        // ...or...
        // Get the run-time type information and process the
        // objects accordingly.
    }
}

```

A pseudocode example of how the memory streaming might be used in a net-working application follows:

```

// Server code:
Stream kOutStream;
// ...insert objects into kOutStream...
int iSize;
char* acBuffer;
kOutStream.Save(acBuffer,iSize);
create begin_stream packet [send iSize];
send packet;
while (not done sending bytes from acBuffer)
{
    create a packet of bytes from acBuffer;
    send packet;
}
create end_stream packet;
send packet;
delete[] acBuffer;

// Client code (in idle loop):
if (received begin_stream packet)
{
    int iSize;
    get iSize from packet;
    char* acBuffer = new char[iSize];
    while (not end_stream packet)
    {
        get packet;
        extract bytes into acBuffer;
    }
}

```



```

        Stream kInStream;
        kInStream.Load(acBuffer,iSize);
        delete[] acBuffer;
        // ...get objects from kInStream and process...
    }

```

18.6.2 THE OBJECT API

The class `Object` has the following API to support streaming:

```

typedef Object* (*FactoryFunction)(Stream&);

class Object
{
public:
    static stdext::hash_map<std::string,FactoryFunction>* ms_pkFactory;
    static bool RegisterFactory ();
    static void InitializeFactory ();
    static void TerminateFactory ();
    static Object* Factory (Stream& rkStream);
    virtual void Load (Stream& rkStream, Stream::Link* pkLink);
    virtual void Link (Stream& rkStream, Stream::Link* pkLink);
    virtual bool Register (Stream& rkStream) const;
    virtual void Save (Stream& rkStream) const;
    virtual int GetDiskUsed (const StreamVersion& rkVersion) const;
}

```

The factory hash map stores class-static functions that are used to load an object from disk. The key of the hash item is the RTTI string. The value is the factory function for the class. For example, class `Object` has the factory function `Object* Factory (Stream&)`. The factory hash table must be created and the factory functions must be added to it during the initialization phase of the application (see Section 18.8). The functions `RegisterFactory` and `InitializeFactory` are built to do this. On termination of the application, the factory hash map must be destroyed. The function `TerminateFactory` does this. The functions `Register`, `Save`, and `GetDiskUsed` are used for saving objects. The functions `Factory`, `Load`, and `Link` are used for loading objects. Each derived class has the same API minus the static hash table and the `TerminateFactory` function. The streaming functions are described in detail here.

Saving a Scene Graph

To save a scene graph, a unique list of objects must be created first. This list is built by a depth-first traversal of the scene. Each `Object` that is visited is told to *register* itself

if it has not already done so. The virtual function that supports this is `Register`. The base class registration function is

```
bool Object::Register (Stream& rkStream) const
{
    Object* pkThis = (Object*)this; // conceptual constness
    if (rkStream.InsertInMap(pkThis,0))
    {
        // used to ensure the objects are saved in the order
        // corresponding to a depth-first traversal of the scene
        rkStream.InsertInOrdered(pkThis);

        for (int i = 0; i < (int)m_kControllers.size(); i++)
        {
            if (m_kControllers[i])
            {
                m_kControllers[i]->Register(rkStream);
            }
        }

        return true;
    }

    return false;
}
```

The stream maintains a hash map of registered objects. The base class `Object` implements this function to ask the stream if the object has been registered. If so, the function returns `false`. If not, the stream adds the object to the hash map, and the object tells the stream to register its only `Object*` members, a list of `Controller` objects. The function then returns `true`. Each derived class implements this function. The base class function is called. If the registration is successful, this object is visited for the first time, and it tells each `Object*` member to register itself. The generic structure is

```
bool DerivedClass::Register (Stream& rkStream) const
{
    if (!BaseClass::Register(rkStream))
    {
        return false;
    }
}
```

```

    for each Object-type member do
    {
        if (member)
        {
            member->Register(rkStream);
        }
    }

    return true;
}

```

After the registration phase, the stream has a list of unique Objects. An iteration is made through the list, and each object is told to save itself. The base class virtual function that supports this is

```

void Object::Save (Stream& rkStream) const
{
    WM4_BEGIN_DEBUG_STREAM_SAVE;

    // RTTI name for factory lookup on Load
    rkStream.Write(std::string(GetType().GetName()));

    // address of object for unique ID on Load/Link
    rkStream.Write((Object*)this);

    // name of object
    rkStream.Write(m_kName);

    // link data
    int iQuantity = (int)m_kControllers.size();
    rkStream.Write(iQuantity);
    for (int i = 0; i < iQuantity; i++)
    {
        rkStream.Write(m_kControllers[i]);
    }

    WM4_END_DEBUG_STREAM_SAVE(Object);
}

```

The RTTI name is a string specific to the class. The string for class Object is “Wm4.Object”; however, Object is an abstract base class, so you will not see this name in a scene file. The class Spatial is also abstract and has the name “Wm4.Spatial”; however, objects of class Node can be instantiated so you will see “Wm4.Node” in the scene files. The RTTI name is used by the stream loader to locate the correct

factory function to create an object of that class. The address of the object is written to disk to be used as a unique identifier when loading. That address will not be a valid memory address when loading, so the stream loader has to resolve these addresses with a linking phase. Each object may have a character string name. Such strings are written to disk by saving the length of the string first, followed by the characters of the string. The null terminator is not written. The controller pointers are also memory addresses that are written to disk for unique identification of the objects. When the controllers themselves are written to disk, those same addresses are the ones that occur immediately after the RTTI names are written.

Each derived class implements `Save`. The base class `Save` is called first. Non-Object data is written to disk first, followed by any `Object*` addresses.

```
void DerivedClass::Save (Stream& rkStream) const
{
    WM4_BEGIN_DEBUG_STREAM_SAVE;

    BaseClass::Save(rkStream);
    write non-object data; // "native" data
    write Object* pointers; // "link" data

    WM4_END_DEBUG_STREAM_SAVE(DerivedClass);
}
```

Not all native data needs to be saved. Some data members are *derivable* from other data and are reproduced once the data is fully loaded from disk. Some data members are set by other run-time processes and need not be saved. For example, `Spatial` has an `Object*` member, the pointer to a parent node. When the scene graph is reconstructed during stream loading, that parent pointer is initialized when the spatial object is attached to a parent by a call to an appropriate `Node` member function. Therefore, the parent pointer is not saved to disk. Some native data may be aggregate data in the form of a class—for example, the class `Vector3`. Various template functions are provided in the streaming source files to save such classes based on memory size. The implication is that any such class cannot have virtual functions. Otherwise, the memory size includes the size of the nonstatic class members as well as the size of the implicit virtual function table pointer.

Although a single scene graph is typically written to disk, the stream object allows multiple objects to be written. For example, you might save a scene graph, a set of camera objects, and a set of light objects. The root node of the scene is what you tell the stream object to save. This node is an example of a *top-level object*. Other objects that are contained in the scene graph are automatically saved, but they are not top-level objects. When you load a scene file that contains multiple top-level objects, you need a way of loading the scene and recapturing these objects. Before a top-level object is saved to disk, the string “Top Level” is written first. This allows the loader to easily identify top-level objects.

A brief explanation is in order for the couple of code samples shown. You saw the macros `WM4_BEGIN_DEBUG_STREAM_SAVE` and `WM4_END_DEBUG_STREAM_SAVE(classname)`. I introduced these to help debug the streaming code for new classes that are added to Wild Magic. Each Object-derived class implements a function called `GetDiskUsed`. The function returns the number of bytes that the object will require for storage on disk. The `Stream` class saves a scene graph to a memory block first, then writes the memory block to disk. In order to have a large enough memory block, the `Stream` queries all the unique objects to be streamed by calling `GetDiskUsed` per object. The sum of the numbers is exactly the number of bytes required for the disk operation. During the streaming to the memory block, `Stream` maintains an index to the location in the memory block where the next write should occur. The “begin” macro saves the index before any writes occur, and the “end” macro saves the index after all writes occur. The difference should be exactly the amount reported by `GetDiskUsed` for that object. If the difference is in error, an assertion is fired. The problem is either that you are incorrectly saving the object to disk or that `GetDiskUsed` itself is incorrectly implemented. The firing of the assertion has been enough for me to track down which of the two is the problem.

Loading a Scene Graph

Loading is a more complicated process than saving. Since the pointer values on disk are invalid, each object must be created in memory first, and then filled in with data loaded from disk. Links between objects such as parent-child relationships must be established later. Despite the invalidity of the disk pointer values, they do store information about the abstract graph that is being loaded. The address of each object in memory is associated with a disk pointer value, so the same hash table that was used for storing the unique objects for saving can be reused for tracking the correspondence between the disk pointer values, called *link IDs*, and the actual memory address of the object. Once all objects are in memory and the hash table is complete with the correspondences, the table is iterated as if it were a list, and the link IDs in each object are replaced by the actual memory addresses. This is exactly the concept of resolving addresses that a linker uses when combining object files created by a compiler.

An object is loaded as follows. The stream object knows that the first thing to expect is either the string “Top Level” or an RTTI string. If “Top Level” is read, the loaded object is stored in a set of top-level objects for the application to access. If an RTTI string is read, the stream knows that it needs to create an object of that type from the file data that follows the RTTI string. The RTTI string is used as a key in a hash map that was created premain at program initialization. The value of a hash map entry is a static class function called `Factory`. This function starts the loading process by creating an object of the desired type, and then filling in its member values

by reading the appropriate data from the file. The factory function for instantiable classes is structured as

```
classname* classname::Factory (Stream& rkStream)
{
    classname* pkObject = new classname;
    Stream::Link* pkLink = new Stream::Link(pkObject);
    pkObject->Load(rkStream,pkLink);
    return pkObject;
}
```

The scene file contains a list of unique objects, each storing a unique identifier called a *link ID*. This identifier was the address of the object when it was saved to disk. Any `Object*` members in an object are themselves old addresses, but are now link IDs that refer to objects that are in the scene file. When loading an object, all link IDs must be stored persistently so that they may be resolved later in a linking phase. The second line of the Factory function creates an object to store these links. The link object itself is stored as the value in a hash map entry whose key is the input object to the constructor. The call to `Load` allows the object to read its native data and `Object*` links from disk. The link object is passed down from derived classes to base classes to allow each base class to add any links it loads.

The `Load` function for the base class `Object` does the work of telling the stream to add the link-object pair to the stream's hash map. After that, the object's native data and links are loaded. The function is

```
void Object::Load (Stream& rkStream, Stream::Link* pkLink)
{
    WM4_BEGIN_DEBUG_STREAM_LOAD;

    // Get old address of object; save it for linking phase.
    Object* pkLinkID;
    rkStream.Read(pkLinkID);
    rkStream.InsertInMap(pkLinkID,pkLink);

    // name of object
    rkStream.Read(m_kName);

    // link data
    int iQuantity;
    rkStream.Read(iQuantity);
    m_kControllers.resize(iQuantity);
    for (int i = 0; i < iQuantity; i++)
```

```

    {
        Object* pkObject;
        rkStream.Read(pkObject);
        pkLink->Add(pkObject);
    }

    WM4_END_DEBUG_STREAM_LOAD(Object);
}

```

Notice how the function loads the controller pointers. At the time the object was saved to disk, this value was the memory address for the controller. Now at load time it can only be used as a unique identifier. That value is stored in the link object for the linking phase that occurs after loading.

Derived classes implement the Load function by calling the base class Load first, and then reading native data followed by link data. This is done in the same order that Save processed the data.

```

void DerivedClass::Load (Stream& rkStream, Stream::Link* pkLink)
{
    WM4_BEGIN_DEBUG_STREAM_LOAD;

    BaseClass::Load(rkStream,pkLink);
    read non-object data; // "native" data
    read Object* pointers; // "link" data
    add Object* pointers to pkLink; // for later linking phase

    WM4_END_DEBUG_STREAM_LOAD(DerivedClass);
}

```

Once all objects are loaded from disk, the linking phase is initiated. An iteration is made over the list of loaded objects and the link function is called for each object. The base class linking function is

```

void Object::Link (Stream& rkStream, Stream::Link* pkLink)
{
    for (int i = 0; i < (int)m_kControllers.size(); i++)
    {
        Object* pkLinkID = pkLink->GetLinkID();
        m_kControllers[i] = (Controller*)rkStream.GetFromMap(pkLinkID);
    }
}

```

The generic structure of the linking function is

```

void classname::Link (Stream& rkStream, Stream::Link* pkLink)
{
    Object* pkLinkID;

    // link member 1
    pkLinkID = GetLinkID();
    m_spkObjectMember1 =
        (ObjectMember1Class*)rkStream.GetFromMap(pkLinkID);

    // ... other object member linking ...

    // link member N
    pkLinkID = GetLinkID();
    m_spkObjectMemberN =
        (ObjectMemberNClass*)rkStream.GetFromMap(pkLinkID);

    // Postlink semantics, if any, go here.
}

```

The function `GetLinkID` accesses a link ID and internally increments a counter so that the next call accesses the next link ID. The objects *must* be linked in the order in which they were saved to disk (which is the same order that they were loaded from disk).

Again, a brief explanation is in order for the couple of code samples shown. You saw the macros `WM4_BEGIN_DEBUG_STREAM_LOAD` and `WM4_END_DEBUG_STREAM_LOAD(classname)`. These are analogous to the macros used for saving a scene. They allow you to track down any implementation errors in streaming when adding new classes to Wild Magic. The `Stream` class loads a scene graph to a memory block first, and then writes the memory block to a scene in memory. In order to have a large enough memory block, the `Stream` queries all the unique objects to be streamed by calling `GetDiskUsed` per object. The sum of the numbers is exactly the number of bytes required for the disk operation. During the streaming to the memory block, `Stream` maintains an index to the location in the memory block where the next read should occur. The “begin” macro saves the index before any reads occur, and the “end” macro saves the index after all reads occur. The difference should be exactly the amount reported by `GetDiskUsed` for that object. If the difference is in error, an assertion is fired. The problem is either that you are incorrectly loading the object from disk or that `GetDiskUsed` itself is incorrectly implemented.

18.7 NAMES AND UNIQUE IDENTIFIERS

Searching for specific objects at run time is useful. The graphics engine supports searching based on a character-string name and on a unique integer-valued identifier.

18.7.1 NAME STRING

An application might require finding objects in the system during run time. To facilitate this, each object has a character string member. The string can be something as simple as a human-readable name, but it could also contain additional information that is useful to the application. For example, the root node of a model of a table could be assigned the name string “Table 17” to identify that the model is in fact a table, with the number indicating that other tables (or types of tables) exist in the scene. It might be important for the application to know what room contains the table. The name string can contain such information, for example, “Table 17 : Room 23.”

To support name strings, the `Object` class provides the following API:

```
public:
    void SetName (const std::string& rkName);
    const std::string& GetName () const;
    static unsigned int GetNextID ();
    virtual Object* GetObjectByName (const std::string& rkName);
    virtual void GetAllObjectsByName (const std::string& rkName,
        std::vector<Object*>& rkObjects);
private:
    std::string m_kName;
```

The member functions `SetName` and `GetName` are standard accessors to the name string. The member function `GetObjectByName` is a search facility that returns a pointer to an object with the specified name. If the caller object has the specified name, the object just returns a pointer to itself. If it does not have the input name, a search is applied to member objects. The method of search depends on the `Object`-derived class itself. Class `Object` compares the input name to the name of the object itself. If found, the object pointer is returned. If not, a search is made over all the controllers attached to the object and, if found, the controller pointer is returned. Otherwise, a null pointer is returned, indicating that an object with the specified name was not found in the current object. A derived-class implementation must call the base class function before checking its own object members.

The name string is not necessarily unique. If two objects have the same name, `GetObjectByName` will find one of them and return a pointer to it. The other object is not found. The other name string member function handles multiple occurrences of a name string. A call to `GetAllObjectsByName` will search for all objects with the input name. The method of search depends on the `Object`-derived class itself.

18.7.2 UNIQUE IDENTIFICATION

Although names are readable and of use to a human, another form of identification may be used to track objects in the system. At first glance you might choose to use

the memory address of the object as a unique identifier since, at a single instant in time, the address is unique. Over time, however, you can run into problems with this scheme. If the memory address of an object is stored by the application to be processed at a later time, it is possible that the object is deleted at some intermediate time. At deletion time the application then has a dangling pointer since the object no longer exists. Worse, other memory allocations can occur with the chance that an entirely new object has the same memory address as the old one that is now defunct. The application no longer has a dangling pointer, but that pointer does not point to the object that the application thinks it is. The likelihood of such an occurrence is higher than you think, especially when the memory manager is asked to allocate and deallocate a collection of homogeneous objects that are all the same size in memory.

To avoid such problems, each object stores a unique identifier. Wild Magic currently uses a 32-bit unsigned integer. The `Object` class has a static unsigned integer member that stores the next available identifier. Each time an object is created, the current static member value is assigned to the nonstatic object member; the static member is then incremented. Hopefully, 32 bits is large enough to provide unique identifiers for all objects over the lifetime of the application. If you have an application that requires more than 2^{32} objects, you can either allow the wraparound that will occur when incrementing the static member, or implement a “counter” class that allows for more bits and provides the simple services of storing a static “next available” counter and incrementing a counter.

To support unique identifiers, the `Object` class provides the following API:

```
public:
    unsigned int GetID () const;
    static unsigned int GetNextID ();
    virtual Object* GetObjectByID (unsigned int uiID);
private:
    unsigned int m_uiID;
    static unsigned int ms_uiNextID;
```

The static member is initialized (premain) to zero. Each constructor for the class has the line of code

```
m_uiID = ms_uiNextID++;
```

This is a simple system that is not designed to reuse an old identifier when an object is deleted. A more sophisticated system could allow reuse, but I believe the additional run-time costs are not warranted.

The member function `GetObjectByID` is similar in structure to the function `GetObjectByName`, except that identifiers are compared rather than name strings. Since the identifiers are unique, there is no need for a function `GetAllObjectsByID`. As with the other search functions, the method of search in an `Object`-derived class is specific to that class.

18.8 INITIALIZATION AND TERMINATION

A class in the object system might declare one or more static members. These members are initialized in the source file for the class. If a static member is itself a class object, the initialization is in the form of a constructor call. This call occurs before the application's main function starts (premain). The destructor is called after the application's main function terminates (postmain). The C++ compiler automatically generates the code for these function calls.

18.8.1 POTENTIAL PROBLEMS

The premain and postmain mechanism has a few potential pitfalls. One problem is that the order in which the function calls occur is unpredictable and is dependent on the compiler. Obtaining a specific order requires some additional coding to force it to occur. Without the forced ordering, one premain initialization might try to use a static object that has not yet been initialized. For example,

```
// contents of Matrix2.h
class Matrix2
{
public:
    Matrix2 (float fE00, float fE01, float fE10, float fE11);
    Matrix2 operator* (float fScalar);
    static Matrix2 IDENTITY;
protected:
    float m_aafE[2][2];
};

// contents of Matrix2.cpp
Matrix2 Matrix2::IDENTITY(1.0f,0.0f,0.0f,1.0f);
Matrix2::Matrix2 (float fE00, float fE01, float fE10, float fE11)
{
    m_aafE[0][0] = fE00; m_aafE[0][1] = fE01;
    m_aafE[1][0] = fE10; m_aafE[1][1] = fE11;
}
// ... other member functions here ...

// contents of MyClass.h
class MyClass
{
public:
    static Matrix2 TWICE_IDENTITY;
};
```

```
// contents of MyClass.cpp
Matrix2 Matrix2::TWICE_IDENTITY = Matrix2::IDENTITY*2.0f;
```

If the static matrix of `MyClass` is initialized first, the static matrix of `Matrix2` has all zero entries, since the storage is reserved already by the compiler but is set to zero values as is all static data.

Problems can occur with file-static data. If a file-static pointer is required to allocate memory, this occurs before the main application is launched. However, since such an initialization is C-style and not part of class semantics, code is not generated to deallocate the memory. For example,

```
int* g_aiData = new int[17];
int main ()
{
    memset(g_aiData,0,17*sizeof(int));
    return 0;
}
```

The global variable `g_aiData` is allocated premain, but no deallocation occurs, thus creating a memory leak. One mechanism to handle this is the `atexit` function provided by C or C++ run-time libraries. The input to `atexit` is a function that takes no parameters and returns void. The functions are executed before the main application exits, but before any global static data is processed postmain. There is an order in this scheme, which is LIFO (last in, first out). The previous block of code can be modified to use this:

```
int* g_aiData = new int[17];
void DeleteData () { delete[] g_aiData; }
int main ()
{
    atexit(DeleteData);
    memset(g_aiData,0,17*sizeof(int));
    return 0;
}
```

Of course in this example the global array is allocated in the same file that has `main`, so a `delete` statement may instead be used before the return from the function. However, if the global array occurs in a different file, then some function in that file has the responsibility for calling `atexit` with the appropriate deletion function as input. That function will be called before `main` returns.

Another problem with file-static data may occur, but it depends on the compiler you use. In order to initialize some part of the system, it might be desirable to force a C-style function to be called premain. The following code in a source file has that effect:

```

bool g_bInitialized = SomeInitialization();
static bool SomeInitialization ()
{
    // Do the initialization.
    return true;
}

```

The initialization function is static because its only purpose is to force something to happen specifically to items in that file. The fact that `g_bInitialized` is global requires the compiler to make the symbol externally accessible by adding the appropriate label to the compiled code in the object file. The compiler should then add the call of the initialization function to its list of such functions to be called premain.

A drawback with this mechanism is that, in fact, the variable `g_bInitialized` is externally accessible. As such, you might have name clashes with symbols in other files. One solution is to create a name for the dummy variable that has a large probability of not clashing with other names. Another solution is to make the dummy variable file-static:

```

static bool gs_bInitialized = SomeInitialization();
static bool SomeInitialization ()
{
    // Do the initialization.
    return true;
}

```

The problem, though, is that an optimizing compiler or a smart linker might try to be too smart. Noticing that `gs_bInitialized` is never referenced anywhere else in the file, and noticing that it is in fact static, the compiler or linker might very well discard the symbol and never add the initialization function to its list of pre-main initializers to call. Yes, this has happened in my experience, and it is a difficult problem to diagnose. A compiler might provide a macro that lets you prevent the static variable from being discarded, but then again it might not. A more robust solution to prevent the discard is

```

static bool gs_bInitialized = SomeInitialization();
static bool SomeInitialization ()
{
    // Do the initialization.
    gs_bInitialized = true;
    return gs_bInitialized;
}

```

Hopefully, the compiler or linker will not try to be really smart and simply notice that the static variable is used somewhere in the file and not discard it. If for some strange

reason the compiler or linker does figure this one out and discards the variable, a more sophisticated body can be used.

To handle order dependencies in the generic solution for classes, discussed in the next section, it is necessary to guard against multiple initializations. The following will do this:

```
static bool gs_bInitialized = SomeInitialization();
static bool SomeInitialization ()
{
    if ( !gs_bInitialized )
    {
        // Do the initialization.
        gs_bInitialized = true;
    }
    return gs_bInitialized;
}
```

The C++ language guarantees that the static data `gs_bInitialized` is zero (false) before any dynamic initialization occurs (the call to `SomeInitialization`), so this code will work as planned to initialize once and only once.

18.8.2 A GENERIC SOLUTION FOR CLASSES

Here is a system that allows a form of premain initialization and postmain termination that takes care of order dependencies. The idea is to register a set of initialization functions and a set of termination functions, all registered premain using the file-static mechanism discussed previously. The initialization and termination functions themselves are called in the main application and make calls to functions that an application is required to implement.

The class `Main` provides the ability to add initialization and termination functions. The class has a member function that executes all the initializers and a member function that executes all the terminators. The initializers and terminators, if any, are called only once. The class structure is

```
class Main
{
public:
    typedef void (*Initializer)(void);
    typedef std::vector<Initializer> InitializerArray;
    static void AddInitializer (Initializer oInitialize);
    static void Initialize ();
```

```

typedef void (*Terminator)(void);
typedef std::vector<Terminator> TerminatorArray;
static void AddTerminator (Terminator oTerminate);
static void Terminate ();

private:
    static InitializerArray* ms_pkInitializers;
    static TerminatorArray* ms_pkTerminators;
    static int ms_iStartObjects;
    static int ms_iFinalObjects;
};

```

The arrays of function pointers are initialized to NULL. The static data members `ms_iStartObjects` and `ms_iFinalObjects` are used to trap object leaks in the program execution. The function to add an initializer is

```

void Main::AddInitializer (Initializer oInitialize)
{
    if (!ms_pkInitializers)
    {
        ms_pkInitializers = new InitializerArray;
    }
    ms_pkInitializers->push_back(oInitialize);
}

```

The initializer array is allocated only if there is at least one initializer. Once all initializers are added to the array, the function `Initialize` is called. Its implementation is shown in the following. Notice the code blocks that are used for detecting object leaks.

```

void Main::Initialize ()
{
    bool bCountIsCorrect = true;

    // Objects should not be created premain.
    if (Object::InUse)
    {
        bCountIsCorrect = false;
        Object::PrintInUse("AppLog.txt",
            "Objects were created before premain initialization");
    }
    assert(bCountIsCorrect);
}

```

```

    if (ms_pkInitializers)
    {
        for (int i = 0; i < (int)ms_pkInitializers->size(); i++)
        {
            (*ms_pkInitializers)[i]();
        }
    }

    delete ms_pkInitializers;
    ms_pkInitializers = 0;

    // number of objects created during initialization
    if (Object::InUse)
    {
        ms_iStartObjects = (int)Object::InUse->size();
    }
    else
    {
        ms_iStartObjects = 0;
    }
}

```

The first time the function is called, the initializers are executed. Afterward, the array of functions is deallocated so that no work must be done in a postmain fashion to free up the memory used by the array. The termination system is identical in structure:

```

void Main::AddTerminator (Terminator oTerminate)
{
    if (!ms_pkTerminators)
    {
        ms_pkTerminators = new TerminatorArray;
    }
    ms_pkTerminators->push_back(oTerminate);
}

void Main::Terminate ()
{
    bool bCountIsCorrect = true;

    // All objects created during the application should be
    // deleted by now.
    if (Object::InUse)

```



```

    {
        ms_iFinalObjects = (int)Object::InUse->size();
    }
    else
    {
        ms_iFinalObjects = 0;
    }

    if (ms_iStartObjects != ms_iFinalObjects)
    {
        bCountIsCorrect = false;
        Object::PrintInUse("AppLog.txt",
            "Not all objects were deleted before postmain termination");
    }

    if (ms_pkTerminators)
    {
        for (int i = 0; i < (int)ms_pkTerminators->size(); i++)
        {
            (*ms_pkTerminators)[i]();
        }
    }

    delete ms_pkTerminators;
    ms_pkTerminators = 0;

    if (bCountIsCorrect)
    {
        // Objects should not be deleted postmain.
        if (Object::InUse)
        {
            ms_iFinalObjects = (int)Object::InUse->size();
        }
        else
        {
            ms_iFinalObjects = 0;
        }

        if (ms_iFinalObjects != 0)
        {
            bCountIsCorrect = false;
            Object::PrintInUse("AppLog.txt",
                "Objects were deleted after postmain termination");
        }
    }
}

```

```

assert(bCountIsCorrect);

// Now that the object leak detection system has completed its tasks,
// delete the hash table to free up memory so that the debug memory
// system will not flag it as a leak.
WM4_DELETE Object::InUse;
Object::InUse = 0;
}

```

Once again I have added code blocks to detect object leaks. If you reach one of the assert statements, you can ignore it and continue the program execution. This will result in an ASCII file written to disk that contains a list of the objects that should have been deleted, but were not. The list includes the unique identifiers stored in the Object class and the object types. This allows you to set break points in the next run to determine why the objects were not deleted. You will find in most cases that the application termination function did not set various smart pointers to null.

For a 2D or 3D graphics application, the Application interface described in Section 18.9 makes use of the initialization and termination scheme described here. The application library effectively provides the following code block:

```

int main (int iQuantity, char** apcArgument)
{
    Main::Initialize();
    int iExitCode = Application::Run(iQuantity,apcArgument);
    Main::Terminate();
    return iExitCode;
}

```

The details of how you hook your application into Application::Run will be discussed in Section 18.9.

Each class requiring initialization services must contain the following in the class definition in the header file:

```

class MyClass
{
public:
    static bool RegisterInitialize ();
    static void Initialize ();
private:
    static bool ms_bInitializeRegistered;
};

```

The source file contains

```
bool MyClass::ms_bInitializeRegistered = false;
bool MyClass::RegisterInitialize ()
{
    if (!ms_bInitializeRegistered)
    {
        Main::AddInitializer(classname::Initialize);
        ms_bInitializeRegistered = true;
    }
    return ms_bInitializeRegistered;
}

void MyClass::Initialize () { <initializations go here> }
```

The registration uses the file-static, premain initialization scheme discussed previously. Similar constructs are used if the class requires termination services.

I have provided macros for the previously mentioned code blocks:

```
WM4_DECLARE_INITIALIZE
WM4_IMPLEMENT_INITIALIZE(classname)
```

The macros are defined in `Wm4Main.mcr`. They may be used if no order dependencies exist for the initialization. If there are dependencies, here is an example of how to handle them. Suppose that Class A initializes some static data and Class B needs that data in order to initialize its own static data. The initializer for A must be called before the initializer for B. The registration function for B is

```
bool B::RegisterInitialize ()
{
    if (!ms_bInitializeRegistered)
    {
        A::RegisterInitialize();
        Main::AddInitializer(B::Initialize);
        ms_bInitializeRegistered = true;
    }
    return ms_bInitializeRegistered;
}
```

This guarantees that the initializer for A occurs in the array of functions *before* the initializer for B. Since the array of functions is executed in the order stored, the correct order of initialization is obtained.

18.9 AN APPLICATION LAYER

At its highest level, a software program needs an application framework in which to run. For a graphics application, we minimally need to create a window and a renderer to draw to it. Game applications as well as some other applications will need input from devices attached to the system. These devices can include a keyboard, a mouse, a gamepad, and/or a joystick. An application must be prepared to handle events generated by these devices. The application might also be networked. Finally, sound is an important part of games. The application can use 2D sound capabilities to play music and special sound effects, but it also might use 3D sound capabilities, taking advantage of 3D sound hardware.

How you structure your application framework depends heavily on what your goals are. If your game will run full-screen on a PC or will run on dedicated hardware such as a game console, you probably will implement an application layer that has support only for what you need.

The Wild Magic application layer is designed to be general purpose and portable. My main goal is for people to purchase my books and run the source code and sample applications on any desktop machine of their choosing, including Microsoft Windows PCs, Macintoshes, and Linux-based PCs. My application layer encapsulates basic functions needed by all platforms and exposes them through an abstract API that is independent of platform. On the back end, I have rudimentary implementations per platform to provide the basic functions. The set of features I support is by no means complete. No doubt you will roll your own layer, so think of mine for what it is—portable and intended to be used in a development environment, not for deployment.

In this section I will describe some of the basic features of my application layer, but most of the concepts apply in one form or another for anyone else's layer. My application subsystem has the following features:

- The initialization and termination of objects via registered functions. This mechanism was described in detail in Section 18.8.
- A console application layer for those applications requiring neither a window nor a renderer. For example, the ScenePrinter tool is an application on the CD-ROM that traverses a scene graph and creates an ASCII file of information about it.
- A window application layer that supports both 2D and 3D applications. Derived classes are provided for the 2D and 3D window applications. The 2D layer has only a renderer for drawing to a bitmap that is later sent to the graphics card to be used as the entire screen. The 3D layer has a camera, as well as a renderer, and is the basis for nearly all the sample applications that are on the CD-ROM.
- An application library that supports both console and windowed applications, so you need only link in one library for any application, regardless of its type.

The application types in Wild Magic version 4 are `ConsoleApplication`, `WindowApplication2`, and `WindowApplication3`. Your application will be a class derived from one of these.

18.9.1 PROCESSING COMMAND-LINE PARAMETERS

The standard entry point into an application is the function

```
int main (int iQuantity, char** apcArgument)
{
    // iQuantity >= 1 is always true.
    // apcArgument[0] is the name of the executing module.

    // ... Process command-line arguments here ...

    return 0;
}
```

The first parameter is the number of strings that occur in the second parameter, an array of pointers to character strings. The input parameters are optional, so it is okay to call `main()` or `main(int)`. The compiler will correctly parse the statement in all cases. The function actually has a third optional parameter, which is used for passing the environment variables, but I do not deal with those in Wild Magic. Clearly, anyone writing an application that accepts inputs to `main` must be prepared to parse the array of strings.

The age-old approach to processing command-line parameters is represented by Henry Spencer's *getopt* routines [Spe06]. The *getopt* routines are limited in that the option names have to be a single letter. Also, the main routine must contain a loop and a switch statement (of options), which repeatedly fetches an argument and decides which option it is and which action to take. I wrote my own command-line parser, which allows option names of length greater than 1, and which allows you to get an argument anywhere in the main routine. This tends to keep the parameter processing and actions together in a related block of code.

The class is `Command` and has the following interface:

```
class Command
{
public:
    Command (int iQuantity, char** apcArgument);
    Command (char* acCmdline);
    ~Command ();

    int ExcessArguments ();
}
```

```

Command& Min (double dValue);
Command& Max (double dValue);
Command& Inf (double dValue);
Command& Sup (double dValue);

int Boolean (char* acName); // returns existence of option
int Boolean (char* acName, bool& rbValue);
int Integer (char* acName, int& riValue);
int Float (char* acName, float& rfValue);
int Double (char* acName, double& rdValue);
int String (char* acName, char*& racValue);
int Filename (char*& racName);

const char* GetLastError ();

protected:
// constructor support
void Initialize ();

// command-line information
int m_iQuantity; // number of arguments
char** m_apcArgument; // argument list (array)
char* m_acCmdline; // argument list (single)
bool* m_abUsed; // arguments already processed

// parameters for bounds checking
double m_dSmall; // bound for argument (min or inf)
double m_dLarge; // bound for argument (max or sup)
bool m_bMinSet; // if true, compare: small <= arg
bool m_bMaxSet; // if true, compare: arg <= large
bool m_bInfSet; // if true, compare: small < arg
bool m_bSupSet; // if true, compare: arg < large

// last error strings
const char* m_acLastError;
static char ms_acOptionNotFound[];
static char ms_acArgumentRequired[];
static char ms_acArgumentOutOfRange[];
static char ms_acFilenameNotFound[];
};

```

The constructor `Command(int, char**)` takes as input the arguments to routine `main`. In a Microsoft Windows application, the constructor `Command(char*)` takes as input the command-line string to `WinMain`. I have designed the Wild Magic version

4 application layer to use only `main`, so you have no need for the second form of the constructor.

After all arguments are processed, the method `ExcessArguments` may be called to check for extraneous information on the command line that does not match what was expected by the program. If extra or unknown arguments appear, then the return value is the index within the command-line string of the first such argument.

When parsing options whose arguments are numerical values, it is possible that upper and lower bounds are required on the input. The bounds for input X are set via calls to `Min` ($\min \leq X$), `Max` ($X \leq \max$), `Inf` ($\inf < X$), or `Sup` ($X < \sup$). These methods return `*this` so that a `Command` object can set bounds and acquire input within the same statement. Some examples are shown later in this section.

The supported option types are *Booleans* (the option takes no arguments), *integers*, *reals*, *strings*, or *file names*. Each type has an associated method whose first `char*` parameter is the option name and whose second parameter will be the option argument, if present. The exceptions are the first Boolean method (an option with no argument) and file names (an argument with no option). The return value of each method is the index within the command-line string, or zero if the option did not occur on the command line.

The function `GetLastError` returns information about problems with reading command-line parameters. The errors are “option not found,” “option requires an argument,” “argument out of range,” and “file name not found.” The user has the responsibility for calling `GetLastError`.

A simple example of command-line parsing is the following. Suppose that you have a program for integrating a function $f(x)$ whose domain is the half-open interval $[a, b)$. The function will be specified as a string, and the endpoints of the interval will be specified as real numbers. Let's assume that we want $0 \leq a < b < 1$. Your program will use samples of the function to produce an approximate value for the integral, so you also want to input the number of partition points as an integer. Finally, your program will write information about the integration to a file.

```
#include "Wm4Command.h"
using namespace Wm4;

// usage:
// "integrate [options] outputfile" with options listed below:
// " -a (float)    : left endpoint (a >= 0, default=0.0)"
// " -b (float)    : right endpoint (a < b < 1, default=0.5)"
// " -num (int)    : number of partitions (default=1)"
// " -func (string): expression for f(x)"
// " -debug        : debug information (default=none)"
// " outputfile    : name of file for output information"

int main (int iQuantity, char** apcArgument)
{
```

```

Command kCmd(iQuantity,apcArgument);

// Get left endpoint (0 <= a is required).
double dA = 0.0f;
kCmd.Min(0.0).Double("a",dA);
if ( kCmd.GetLastError() )
{
    cout << "0 <= a required" << endl;
    return 1;
}

// Get right endpoint (a < b < 1 is required).
double dB = 0.5;
kCmd.Inf(dA).Sup(1.0).Double("b",dB);
if ( kCmd.GetLastError() )
{
    cout << "a < b < 1 required" << endl;
    return 2;
}

// Get number of partition points (1 or larger).
int iPoints = 1;
kCmd.Min(1).Integer("num",iPoints);
if ( kCmd.GetLastError() )
{
    cout << "num parameter must be 1 or larger" << endl;
    return 3;
}

// Get function expression (must be supplied).
char acFunction[128];
if ( !kCmd.String("func",acFunction) )
{
    cout << "function must be specified" << endl;
    return 4;
}

// Get output file name.
char acOutfile[128];
if ( !kCmd.Filename(acOutfile) )
{
    cout << "output file must be specified" << endl;
    return 5;
}

```



```

// Want debug information?
bool bDebug = false;
kCmd.Boolean("debug",bDebug);

// Check for extraneous or unknown options.
if ( kCmd.ExcessArguments() )
{
    cout << "command line has excess arguments" << endl;
    return 6;
}

// Your program code goes here.
return 0;
}

```

18.9.2 THE APPLICATION CLASS

The base class of the entire application library is quite simple. It is called `Application` and has the following interface:

```

class Application
{
    WM4_DECLARE_TERMINATE;

public:
    virtual ~Application ();

    static Application* TheApplication;
    static Command* TheCommand;

    typedef int (*EntryPoint)(int, char**);
    static EntryPoint Run;

    void SetExtraData (int iIndex, int iSize,
        const void* pvData);
    void GetExtraData (int iIndex, int iSize,
        void* pvData) const;
    bool LaunchFileDialog () const;

protected:
    Application ();

    enum { APP_EXTRA_DATA_QUANTITY = 128 };
    char m_acExtraData[APP_EXTRA_DATA_QUANTITY];

```

```

    bool m_bLaunchFileDialog;
};

WM4_REGISTER_TERMINATE(Application);

```

The class is abstract since its only constructor is protected. This class contains the minimum support for all the application types: console, 2D windowed, and 3D windowed. The engine is designed to handle a single application; that is, the existence of multiple instances of the same application is unknown to the engine. Because only a single instance of an application is assumed, a pointer to the unique application object is stored in the base class and is named `TheApplication`. The event handlers of the windowed applications are C-style functions that are not member functions in the `Application` class hierarchy. The handlers must be able to pass along events to the application object. They do so through the `TheApplication` pointer. The base class also stores a unique object for the command-line parameters. The uniqueness is clear: You cannot pass two command lines to the same executable module.

The entry point to the application is through the static data member `Run`. The `int` parameter is the number of command-line arguments. The `char**` parameter is the array of argument strings. The final derived classes (your applications) must set this function pointer to an appropriately designed function. The mechanism is described later in this section. A function pointer is used rather than a member function to allow all application types to coexist in the library. If a member function were to be used instead, each application type would have to implement that function, leading to multiply defined functions in the library—an error that the linker will report to you.

In Section 18.8, I mentioned a small code block for the `main` function. The actual source code is in the file `Wm4Application.cpp`. Since `main` is what the compiler expects as the entry point, the function cannot be a class member. The code is

```

int main (int iQuantity, char* apcArgument[])
{
    // Sorry! If you want to use the path to the Wild Magic 4
    // folder for the purpose of searching for data files, you
    // will need to create the WM4_PATH environment variable.
    // The sample Wild Magic applications rely on
    // System::WM4_PATH to find scene graph object files
    // (.wmof), image files (.wmif), and shader program files
    // (.wmsp).
    assert(System::WM4_PATH != std::string(""));

    Main::Initialize();

    int iExitCode = 0;
    if (Application::Run)
    {

```

```

        // Always check the current working directory.
        System::InsertDirectory(".");

        // the path to scene graph files
        std::string kDir;
        kDir = System::WM4_PATH + std::string("/Data/Wmof");
        System::InsertDirectory(kDir.c_str());

        // the path to image files
        kDir = System::WM4_PATH + std::string("/Data/Wmif");
        System::InsertDirectory(kDir.c_str());

        // the path to shader program files
        kDir = System::WM4_PATH + std::string("/Data/Wmsp");
        System::InsertDirectory(kDir.c_str());

        Application::TheCommand = WM4_NEW Command(iQuantity,
            apcArgument);
        iExitCode = Application::Run(iQuantity,apcArgument);
        WM4_DELETE Application::TheCommand;
        Application::TheCommand = 0;

        System::RemoveAllDirectories();
    }
    else
    {
        iExitCode = INT_MAX;
    }

    Main::Terminate();

    WM4_DELETE Application::TheApplication;
    Application::TheApplication = 0;

#ifdef WM4_MEMORY_MANAGER
#ifdef _DEBUG
    Memory::GenerateReport("MemoryReportDebug.txt");
#else
    Memory::GenerateReport("MemoryReportRelease.txt");
#endif
#endif

    return iExitCode;
}

```

Wild Magic version 4 now supports the concept of a user-defined “path,” a list of directories to search for data files. This system requires you to create an environment variable through whatever mechanism your operating system requires. The variable must be named `WM4_PATH` and it must be set to the directory where Wild Magic version 4 was installed. For example, if you installed the CD-ROM contents to

```
X:/GeometricTools/WildMagic4
```

then you should set `WM4_PATH` to this directory. During premain initialization, I read this environment variable and use it for the purpose of creating other directories that are part of the distribution. The first line of `main` is an assertion that tests whether you have set the environment variable.

As discussed previously, all registered initialization functions are executed before the application is run. The application itself is created during the initialization phase—more on this a little bit later. If the application Run function pointer has not been set, the application cannot run. This error will occur if you forget to use the initialization system properly when creating your application classes.

Assuming the application’s Run function is set, the command-line object is created for use by the application, and then the Run function is executed. On completion, the command-line object is destroyed. All registered termination functions are then executed. The goal is to trap object leaks that the application might have.

The deletion of the application object is delayed until the very end of the `main` function, which forces you to correctly clean up any objects in your application when a termination callback is executed. I made this choice so that the graphics system has a chance to release any resources that are associated with the application: textures, shader programs, cached arrays, and anything else you might have cached in VRAM on the graphics card. If you have not freed all your objects, `Main::Terminate` will complain loudly that you forgot to clean up!

18.9.3 THE `ConsoleApplication` CLASS

Console applications do not require a window for displaying results. In a straightforward C or C++ program, you would implement such an application using `main` directly. My application layer supports console applications, but they require more setup than just implementing a single function—a natural consequence of the design of `main` in the `Application` class.

The `ConsoleApplication` class has the interface

```
class ConsoleApplication : public Application
{
public:
    ConsoleApplication ();
    virtual ~ConsoleApplication ();
```

```

        virtual int Main (int iQuantity, char** apcArgument) = 0;

protected:
    static int Run (int iQuantity, char** apcArgument);
};

```

The class is abstract because of the presence of the pure virtual function `Main`. This is not to be confused with the class `Main`. I choose to use the name because, effectively, the function is the entry point into an application that resembles the standard application entry point `int main(int, char**)`. Your applications must implement `Main`.

The `Run` function is

```

int ConsoleApplication::Run (int iQuantity, char** apcArgument)
{
    ConsoleApplication* pkTheApp =
        (ConsoleApplication*)TheApplication;
    return pkTheApp->Main(iQuantity, apcArgument);
}

```

A console application will set the function pointer `Application::Run` to its own `Run` function pointer. When `Application::Run` is executed by the main function in class `Application`, the `ConsoleApplication::Run` will be executed. All that it does is pass on the command-line parameters to the derived class's implementation of `Main`. This is technically not required since `Application::TheCommand` was already constructed in `main`, and the derived class has access to it. However, I did not want to force you to use the command-line object; you can parse the parameters yourself, if you so choose.

The final piece of the puzzle is to derive a class from `ConsoleApplication` and hook up the `Run` function by using the initialization mechanism. An additional macro is provided, in the file `Wm4Application.mcr`, to implement the initialization and create the application object, both without having to type in the code yourself. The macro is `WM4_CONSOLE_APPLICATION` and is defined by

```

#define WM4_CONSOLE_APPLICATION(classname) \
WM4_IMPLEMENT_INITIALIZE(classname); \
\
void classname::Initialize () \
{ \
    Application::Run = &ConsoleApplication::Run; \
    TheApplication = new classname; \
}

```

The initialization function sets the `Application::Run` function pointer so, indeed, your application will be run when you click the “go” button. The second line of code in the initialization function acts as a factory to create an object from your specific application class.

The following example illustrates what you must do for a hypothetical class `MyConsoleApplication`:

```
// in MyConsoleApplication.h

#include "Wm4ConsoleApplication.h"
using namespace Wm4;

class MyConsoleApplication : public ConsoleApplication
{
    WM4_DECLARE_INITIALIZE;
public:
    MyConsoleApplication ();
    virtual ~MyConsoleApplication ();
    virtual int Main (int iQuantity, char** apcArgument);
protected:
    // ... Whatever else you need goes here ...
};

WM4_REGISTER_INITIALIZE(MyConsoleApplication);
```

The declaration macro for initialization is used to indicate your intention to have an `Initialize` function called by `Main::Initialize`. The registration macro generates the code to force the registration of `Initialize` with class `Main`.

The source file has

```
// in MyConsoleApplication.cpp

#include "MyConsoleApplication.h"
using namespace Wm4;

WM4_CONSOLE_APPLICATION(MyConsoleApplication);

int MyConsoleApplication::Main (int iQuantity, char** apcArgument)
{
    // ... Do your thing here ...
    return 0;
}
```

The order of events is as follows:

1. The `MyConsoleApplication::Initialize` function is registered premain; that is, before `int main(int, char**)` executes.
2. `int main(int, char**)` is executed.
3. `Main::Initialize` is called.

4. `MyConsoleApplication::Initialize` is called. The function pointer `ConsoleApplication::Run` is assigned to `Application::Run`. A `MyConsoleApplication` object is dynamically created, and its pointer is assigned to `Application::TheApplication`.
5. `Application::TheCommand` is dynamically created, using the `int` and `char**` parameters that were passed to `int main(int, char**)`.
6. The function that `Application::Run` points to is executed. In this case, it is `ConsoleApplication::Run`, which in turn calls the function `MyConsoleApplication::Main` for your application object.

For the most part, all these details are hidden from you. All you should care about are creating the skeleton class, as shown, and implementing the `Main` function for your particular needs.

18.9.4 THE WINDOWAPPLICATION CLASS

The mechanism for working with windowed applications is similar to that for console applications. The base class for such applications is `WindowApplication` and is the common framework that occurs in 2D and 3D applications. The portion of the interface similar to the console interface is

```
class WindowApplication : public Application
{
public:
    WindowApplication (const char* acWindowTitle, int ixPosition,
                      int iyPosition, int iWidth, int iHeight,
                      const ColorRGB& rkBackgroundColor);
    virtual ~WindowApplication ();

    virtual int Main (int iQuantity, char** apcArgument);

protected:
    static int Run (int iQuantity, char** apcArgument);
};
```

The `Run` function is

```
int WindowApplication::Run (int iQuantity, char** apcArgument)
{
    WindowApplication* pkTheApp = (WindowApplication*)TheApplication;
    return pkTheApp->Main(iQuantity, apcArgument);
}
```

and has exactly the same purpose as that of `ConsoleApplication::Run`.

The mechanism to hook up your application to be run is supported by the macro

```
#define WM4_WINDOW_APPLICATION(classname) \
WM4_IMPLEMENT_INITIALIZE(classname); \
\
void classname::Initialize () \
{ \
    Application::Run = &WindowApplication::Run; \
    TheApplication = new classname; \
}
```

The macro is structured exactly the same as the `WM4_CONSOLE_APPLICATION` macro. Unlike the console applications, an additional layer occurs between your application and the `WindowApplication` class. The derived class `WindowApplication2` supports 2D applications; the derived class `WindowApplication3` supports 3D applications.

An example to illustrate setting up a 3D application uses a hypothetical class `MyWindowApplication`:

```
// in MyWindowApplication.h

#include "Wm4WindowApplication3.h"
using namespace Wm4;

class MyWindowApplication : public WindowApplication3
{
    WM4_DECLARE_INITIALIZE;
public:
    MyWindowApplication ();
    virtual ~MyWindowApplication ();

    // ... Other interface functions go here ...
protected:
    // ... Whatever else you need goes here ...
};

WM4_REGISTER_INITIALIZE(MyWindowApplication);
```

The source file has

```
// in MyWindowApplication.cpp

#include "MyWindowApplication.h"
using namespace Wm4;
```



```

WM4_WINDOW_APPLICATION(MyWindowApplication);

MyWindowApplication::MyWindowApplication ()
:
    WindowApplication3("MyWindowApplication",0,0,640,480,
        ColorRGBA::WHITE)
{
    // ... Initializations go here ...
}

// ... Your class implementation goes here ...

```

The `WindowApplication::Main` is implemented by `WindowApplication`, in comparison to `ConsoleApplication::Main`, which required the override to occur in the final application class. The implementations of `WindowApplication::Main` are platform-specific because the windowing systems and event handling are platform-specific. More about this later.

Construction

The `WindowApplication` class and its derivations all have a constructor of the form

```

class WindowApplication : public Application
{
public:
    WindowApplication (const char* acWindowTitle, int ixPosition,
        int iyPosition, int iWidth, int iHeight,
        const ColorRGBA& rkBackgroundColor);
};

```

The window title is intended to be displayed on the title bar of the window. The position parameters are the location on the screen of the upper-left corner of the window; the width and height are the size of the window; and the input color is used for clearing the background by setting all pixels to that color. The final application class always declares the default constructor whose implementation calls the base class constructor with the appropriate parameters.

A portion of the `WindowApplication` is devoted to the access of the members set by the constructor:

```

class WindowApplication : public Application
{
public:
    const char* GetWindowTitle () const;
    int GetXPosition () const;

```

```

    int GetYPosition () const;
    int GetWidth () const;
    int GetHeight () const;
    void SetRenderer (Renderer* pkRenderer);
    void SetWindowID (int iWindowID);
    int GetWindowID () const;

protected:
    const char* m_acWindowTitle;
    int m_iXPosition, m_iYPosition, m_iWidth, m_iHeight;
    ColorRGBA m_kBackgroundColor;
    int m_iWindowID;
    Renderer* m_pkRenderer;
};

```

The Get routines have the obvious behavior. A typical windowing system will assign a unique identifier (a window handle) to each window it creates. During the window creation, that identifier must be stored by the window for identification purposes throughout the program run time. The data member `m_iWindowID` stores that value and is assigned by a call to `SetWindowID`. The renderer creation is dependent on the operating system, the windowing system, and the graphics API (OpenGL, Direct3D, or the software renderer). The platform-specific source code will create a renderer, and then pass it to the `WindowApplication` object by calling the function `SetRenderer`. Notice that the renderer is stored polymorphically through the abstract base class `Renderer`—a requirement for the `WindowApplication` interface to be platform-independent.

Event Handling

All windowing systems have mechanisms for handling events such as key presses, mouse clicks and motion, and repositioning and resizing of windows. They also have mechanisms for repainting the screen when necessary and for idle-time processing when the event queue is empty. The class `WindowApplication` has a collection of *event callbacks*—functions that are called by the platform-specific implementations of the event handlers and dispatchers. These callbacks are

```

class WindowApplication : public Application
{
public:
    virtual bool OnPrecreate ();
    virtual bool OnInitialize ();
    virtual void OnTerminate ();
    virtual void OnMove (int iX, int iY);
    virtual void OnResize (int iWidth, int iHeight);

```

```

virtual void OnDisplay ();
virtual void OnIdle ();
virtual bool OnKeyDown (unsigned char ucKey, int iX, int iY);
virtual bool OnKeyUp (unsigned char ucKey, int iX, int iY);
virtual bool OnSpecialKeyDown (int iKey, int iX, int iY);
virtual bool OnSpecialKeyUp (int iKey, int iX, int iY);
virtual bool OnMouseClicked (int iButton, int iState, int iX,
    int iY, unsigned int uiModifiers);
virtual bool OnMotion (int iButton, int iX, int iY);
virtual bool OnPassiveMotion (int iX, int iY);

void RequestTermination ();
};

```

The typical structure of the main function in a windowing system is the following pseudocode:

```

int WindowApplication::Main (...)
{
    (1) Do work if necessary before window creation;
    (2) Create the window;
    (3) Create the renderer;
    (4) Initialize the application;
    (5) Display the window;
    do_forever
    {
        if ( message pending )
        {
            (6) If message is to quit, break out of loop;
            (7) Dispatch the message;
        }
        else
        {
            (8) Do idle processing;
        }
    }
    (9) Terminate the application;
}

```

Naturally, this function runs forever until a message is sent for the application to quit. The event callbacks are executed directly, or indirectly, during this function call. The callback `OnPrecreate` is called during stage (1). The window creation (2) is specific to the windowing system used by the operating system. The renderer creation (3) is specific to the graphics API. Stage (4) is managed by the callback `OnInitialize`. The window display (5) is part of the windowing API and is not part

of my application library. The message pump is the `do_forever` loop. If the quit message is received, the loop is exited and the application terminates. The application is given a chance to clean up at stage (9) via the callback `OnTerminate`. The decision for an application to terminate can be implicit in the architecture (for example, when a user clicks on the window “close” button) or explicit (for example, when a user presses a specified key). In my applications, the default key is ESC. The function `RequestTermination` is called within my application library code and generates a quit message. The implementation is platform-specific. For example, the Microsoft Windows API function `PostMessage` is called to post a `WM_DESTROY` message to the message queue.

Stage (7) is where the events are dispatched to an event handler. In my application architecture, the handler accesses the application object through the `Application::TheApplication` pointer, determines the type of the event that has occurred, and then tells the application object to execute its corresponding callback. Window translation generates an event that causes `OnMove` to be called. Window resizing generates an event that causes `OnResize` to be called. If a window is partially covered or minimized, and then uncovered or maximized, the window must be repainted (in part or in full). This type of event causes `OnDisplay` to be called.

Key presses are events that cause the functions `OnKeyDown`, `OnKeyUp`, `OnSpecialKeyDown`, and `OnSpecialKeyUp` to be called. The special keys are the arrow keys; the insert, delete, home, end, page up, and page down keys; and the function keys, F1 through F12. The callback `OnSpecialKeyDown` is executed when one of these keys is pressed. The callback `OnSpecialKeyUp` is executed when the key is released. The remaining keys on the keyboard are handled similarly by `OnKeyDown` and `OnKeyUp`.

The key identifiers tend to be constants provided by the platform’s windowing system, and their values are not consistent across platforms—another source of non-portability. `WindowApplication` has a collection of `const` data members that are assigned the key identifiers in each platform-dependent implementation. These data members provide a consistent naming convention for your applications so that they may remain portable. For example, a few of these data members are

```
class WindowApplication : public Application
{
public:
    // keyboard identifiers
    static const int KEY_ESCAPE;
    static const int KEY_LEFT_ARROW;
    // ... other const data members ...

    // keyboard modifiers
    static const int KEY_SHIFT;
    static const int KEY_CONTROL;
    // ... other const data members ...
};
```

Mouse events include pressing a mouse button and moving the mouse, which generates calls to `OnClick`, `OnMotion`, and `OnPassiveMotion`. If your development platform is Microsoft Windows or X-Windows, you might have been tempted to have more mouse callbacks such as `OnLeftMouseDown` and `OnMiddleMouseUp`, which is reasonable for hardware and operating systems that support multiple-button mice. However, a Macintosh mouse has only a single button, so I refrained from having anything other than `OnClick`. I do pass in the button type (`iButton`), a button state (`iState`), and button modifiers (`uiModifiers`). The platform-independent names I use for these are

```
class WindowApplication : public Application
{
public:
    // mouse buttons
    static const int MOUSE_LEFT_BUTTON;
    static const int MOUSE_MIDDLE_BUTTON;
    static const int MOUSE_RIGHT_BUTTON;

    // mouse state
    static const int MOUSE_UP;
    static const int MOUSE_DOWN;

    // mouse modifiers
    static const int MOD_LBUTTON;
    static const int MOD_MBUTTON;
    static const int MOD_RBUTTON;
};
```

So in fact, you can write application code that works fine under Microsoft Windows and X-Windows, but not on the Macintosh. My advice is to use only the left mouse button `MOUSE_LEFT_BUTTON` for applications you intend to be portable. Alternatively, you can rewrite the Macintosh application code in a manner that maps combinations of the mouse button and modifiers to simulate a three-button mouse.

Mouse motion is handled in one of two ways. For mouse dragging, the idea is to detect that the mouse is moving while one of the mouse buttons is pressed. The callback that is executed in this situation is `OnMotion`. For processing mouse motion when no mouse buttons are pressed, the callback `OnPassiveMotion` is executed.

The `OnPrecreate` and `OnInitialization` callbacks return a Boolean value. In normal situations, the returned value is `true`, indicating that the calls were successful and the application may continue. If an abnormal condition occurs, the returned value is `false`. The application terminates early on such a condition. For example, if your `OnInitialize` function attempts to load a scene graph file, but fails to find that file, the function returns `false` and the application should terminate. Naturally, you should

structure your applications to be successful. But if an abnormal condition occurs, exit gracefully!

The key and mouse callbacks also return Boolean values. If the value is true, the callback has processed the event itself. For example, if your application implements `OnKeyDown` to process the X-key, and the X-key is actually pressed, your callback will detect that key and do something, after which it returns true. If the callback does not do anything when Y is pressed, and the callback receives the Y-key and ignores it, the return value is false. This mechanism gives a final application the ability to determine if my base class key handlers processed the keys, and then choose to ignore that key itself. That said, nothing forces you to call the base class functions. The value you return is ignored by the platform-specific event handlers.

The idle processing is handled by the callback `OnIdle`. The 3D applications make extensive use of this callback in order to achieve real-time frame rates. You might be tempted to use a *system timer* to control the frame rate. The problem, though, is that many system timers have a limited resolution. For example, the `WM_TIMER` event in the Microsoft Windows environment occurs at an approximate rate of 18 times per second. Clearly, this will not support real-time applications.

Finally, a few interface functions in `WindowApplication` support font handling. Recall that the `Renderer` class can be told to use fonts other than the default ones used by the graphics API. If your application will overlay the rendered scene with text, you most likely will need to know font metrics in order to properly position the text. Simple metrics are provided by

```
class WindowApplication : public Application
{
public:
    int GetStringWidth (const char* acText) const;
    int GetCharacterWidth (const char cCharacter) const;
    int GetFontHeight () const;
};
```

The implementations are dependent on the windowing system, so they occur in the source files containing the platform-specific code.

18.9.5 THE WINDOWAPPLICATION3 CLASS

The class that supports the 3D applications is `WindowApplication3` and is derived from `WindowApplication`. Its interface is a bit lengthy:

```
class WindowApplication3 : public WindowApplication
{
public:
    WindowApplication3 (const char* acWindowTitle, int ixPosition,
```

```

        int iYPosition, int iXSize, int iYSize,
        const ColorRGBA& rkBackgroundColor);
virtual ~WindowApplication3 ();

virtual bool OnInitialize ();
virtual void OnTerminate ();
virtual void OnDisplay ();
virtual bool OnKeyDown (unsigned char ucKey, int iX, int iY);
virtual bool OnSpecialKeyDown (int iKey, int iX, int iY);
virtual bool OnSpecialKeyUp (int iKey, int iX, int iY);
virtual bool OnMouseClicked (int iButton, int iState, int iX, int iY,
        unsigned int uiModifiers);
virtual bool OnMotion (int iButton, int iX, int iY);

protected:
    // camera motion
    void InitializeCameraMotion (float fTrnSpeed, float fRotSpeed,
        float fTrnSpeedFactor = 2.0f, float fRotSpeedFactor = 2.0f);
    virtual bool MoveCamera ();
    virtual void MoveForward ();
    virtual void MoveBackward ();
    virtual void MoveUp ();
    virtual void MoveDown ();
    virtual void TurnLeft ();
    virtual void TurnRight ();
    virtual void LookUp ();
    virtual void LookDown ();
    CameraPtr m_spkCamera;
    Vector3f m_akWorldAxis[3];
    float m_fTrnSpeed, m_fTrnSpeedFactor;
    float m_fRotSpeed, m_fRotSpeedFactor;
    bool m_bUArrowPressed, m_bDArrowPressed, m_bLArrowPressed;
    bool m_bRArrowPressed, m_bPgUpPressed, m_bPgDnPressed;
    bool m_bHomePressed, m_bEndPressed, m_bCameraMoveable;

    // object motion
    void InitializeObjectMotion (Spatial* pkMotionObject);
    bool MoveObject ();
    void RotateTrackBall (float fx0, float fy0, float fx1,
        float fy1);
    SpatialPtr m_spkMotionObject;
    int m_iDoRoll, m_iDoYaw, m_iDoPitch;
    float m_fXTrack0, m_fYTrack0, m_fXTrack1, m_fYTrack1;
    Matrix3f m_kSaveRotate;

```

```

bool m_bUseTrackBall, m_bTrackBallDown;
bool m_bObjectMoveable;

// performance measurements
void ResetTime ();
void MeasureTime ();
void UpdateFrameCount ();
void DrawFrameRate (int iX, int iY, const ColorRGBA& rkColor);
double m_dLastTime, m_dAccumulatedTime, m_dFrameRate;
int m_iFrameCount, m_iTimer, m_iMaxTimer;
};

```

Most of the event callbacks are stubbed out in the base class `WindowApplication` to do nothing. The event callbacks in the derived class that have some work to do are listed in the public section of the interface.

The protected section of the interface is decomposed into three subsections. The first subsection contains the declaration of the camera, `m_spkCamera`, to be used by the renderer. Naturally, you can create your own cameras, but the one provided by this class is the one that gets hooked up to various events in order to translate and rotate the camera. The remaining data members in the subsection are all related to handling camera motion. The camera can be moved via the arrow keys and other special keys.

The second subsection is related to object motion; the object is specified by your application—typically the entire scene graph. Only rotations are supported by the application library. Objects can be rotated in two ways. First, you can rotate the object using the function keys F1 through F6. Second, the class has a *virtual trackball* that surrounds the scene. Dragging the mouse with the left button depressed allows you to rotate the trackball.

The third subsection is for performance measurements—specifically for measuring the frame rate of your application.

The following sections discuss each of these topics.

Camera Motion

Given a camera with eye point $\mathbf{E}$, view direction $\mathbf{D}$, up direction $\mathbf{U}$, and right direction $\mathbf{R}$, all in world coordinates, the tendency is to update the position of the eye point and the orientation of the camera *relative to the camera coordinate frame itself*. The operations are summarized here. The new coordinate frame quantities are denoted with prime symbols: $\mathbf{E}'$, $\mathbf{D}'$, $\mathbf{U}'$, and $\mathbf{R}'$.

Let $s > 0$ be the speed of translation. The translation in the view direction causes only the eye point location to change:

$$\mathbf{E}' = \mathbf{E} \pm s\mathbf{D}$$

The translation in the up direction is

$$\mathbf{E}' = \mathbf{E} \pm s\mathbf{U}$$

and the translation in the right direction is

$$\mathbf{E}' = \mathbf{E} \pm s\mathbf{R}$$

Let $\theta > 0$ be an angle of rotation. The corresponding rotations are counterclockwise in the plane perpendicular to the rotation axis, looking down the axis at the plane; the direction you look in is the negative of the axis direction. The eye point is never changed by the rotations. The rotation about the view direction preserves that direction itself, but changes the other two:

$$\begin{bmatrix} \mathbf{U}' \\ \mathbf{R}' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} \mathbf{U} \\ \mathbf{R} \end{bmatrix} \quad (18.1)$$

In the implementation, after the rotation you need to assign the results back to the storage of the vectors; that is, $\mathbf{U} \leftarrow \mathbf{U}'$ and $\mathbf{R} \leftarrow \mathbf{R}'$. The rotation about the up vector is

$$\begin{bmatrix} \mathbf{R}' \\ \mathbf{D}' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} \mathbf{R} \\ \mathbf{D} \end{bmatrix} \quad (18.2)$$

and the rotation about the right vector is

$$\begin{bmatrix} \mathbf{D}' \\ \mathbf{U}' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} \mathbf{D} \\ \mathbf{U} \end{bmatrix} \quad (18.3)$$

As I mentioned, the tendency is for you to want to update the camera frame in this manner. The problem in an application, though, is that you usually have a world coordinate system that has a preferred up direction that remains fixed throughout the application's lifetime. The up vector for the camera changes on a roll about the view direction and on a pitch about the right vector. After a roll, a translation in the camera up direction is not a translation in the world up direction. After a pitch, a translation in the camera view direction is not a translation perpendicular to the world up direction. Consider the situation where the camera represents the viewing system of a character player. If the character walks along a horizontal floor and his view direction is parallel to the floor, any translation of his eye point should keep him on that floor. Now imagine that the character looks down at the floor. His view direction is no longer parallel to the floor. If you were to translate the eye point in the view direction, the character would walk toward the floor (and directly through it). The preferable option would be to have the character walk parallel to the floor, even though he is looking down. This requires using the world coordinate frame for the incremental translations and rotations and applying the transformations to the camera frame.

The array `m_akWorldAxis` stores the world coordinate frame for the purposes of camera motion. The translation speed is `m_fTrnSpeed`, and the rotation speed is `m_fRotSpeed`. The other two float data members in the camera motion section, `m_fTrnSpeedFactor` and `m_fRotSpeedFactor`, are multiplicative (or division) factors for adjusting the current speeds. The member function `InitializeCameraMotion` initializes the speeds and factors. It also uses the camera's *local coordinate axes* at the time of the call to `initialize_m_akWorldAxis`. The camera local axes are presumably set in the application's `OnInitialize` call to place the camera in the world coordinate frame of the scene graph. The entry 0 of the world axis array may be thought of as the view direction in the world. The entry 1 is thought of as the up vector, and the entry 2 is thought of as the right vector. The Boolean member `m_bCameraMoveable` indicates whether or not the camera is set up for motion. By default, the value is `false`. The value is set to `true` when `InitializeCameraMotion` is called.

The member functions `MoveForward` and `MoveBackward` translate the camera frame in the world view direction:

```
void WindowApplication3::MoveForward ()
{
    Vector3f kLoc = m_spkCamera->GetLocation();
    kLoc += m_fTrnSpeed*m_akWorldAxis[0];
    m_spkCamera->SetLocation(kLoc);
}

void WindowApplication3::MoveBackward ()
{
    Vector3f kLoc = m_spkCamera->GetLocation();
    kLoc -= m_fTrnSpeed*m_akWorldAxis[0];
    m_spkCamera->SetLocation(kLoc);
}
```

The translation keeps the camera parallel to the plane perpendicular to the world up vector, even if the observer is looking down at that plane. Similarly, the member functions `MoveUp` and `MoveDown` translate the camera frame in the world up direction:

```
void WindowApplication3::MoveUp ()
{
    Vector3f kLoc = m_spkCamera->GetLocation();
    kLoc += m_fTrnSpeed*m_akWorldAxis[1];
    m_spkCamera->SetLocation(kLoc);
}

void WindowApplication3::MoveBackward ()
{
    Vector3f kLoc = m_spkCamera->GetLocation();
```

```

        kLoc -= m_fTrnSpeed*m_akWorldAxis[1];
        m_spkCamera->SetLocation(kLoc);
    }

```

Note that in the four functions the translations change neither the camera axis directions nor the world axis directions. I do not provide implementations for translating left or right.

The member functions TurnLeft and TurnRight are rotations about the world up vector:

```

void WindowApplication3::TurnLeft ()
{
    Matrix3f kIncr(m_akWorldAxis[1],m_fRotSpeed);
    m_akWorldAxis[0] = kIncr*m_akWorldAxis[0];
    m_akWorldAxis[2] = kIncr*m_akWorldAxis[2];

    Vector3f kDVector = kIncr*m_spkCamera->GetDVector();
    Vector3f kUVector = kIncr*m_spkCamera->GetUVector();
    Vector3f kRVector = kIncr*m_spkCamera->GetRVector();
    m_spkCamera->SetAxes(kDVector,kUVector,kRVector);
}

void WindowApplication3::TurnRight ()
{
    Matrix3f kIncr(m_akWorldAxis[1],-m_fRotSpeed);
    m_akWorldAxis[0] = kIncr*m_akWorldAxis[0];
    m_akWorldAxis[2] = kIncr*m_akWorldAxis[2];

    Vector3f kDVector = kIncr*m_spkCamera->GetDVector();
    Vector3f kUVector = kIncr*m_spkCamera->GetUVector();
    Vector3f kRVector = kIncr*m_spkCamera->GetRVector();
    m_spkCamera->SetAxes(kDVector,kUVector,kRVector);
}

```

The first blocks of code in the functions perform the rotations in Equation (18.2), but applied to the world axis vectors. You might think that only the camera view direction and right vectors need to be updated, but that is only the case when the camera up vector is in the same direction as the world up vector. If the observer is looking down at the floor, a rotation about the world up vector will change *all* the camera axis directions. Thus, the second blocks of code in the functions apply the rotation to all the camera axis directions.

The member functions LookUp and LookDown are pitch rotations about the world's right vector:

```

void WindowApplication3::LookUp ()
{
    Matrix3f kIncr(m_akWorldAxis[2],-m_fRotSpeed);

    Vector3f kDVector = kIncr*m_spkCamera->GetDVector();
    Vector3f kUVector = kIncr*m_spkCamera->GetUVector();
    Vector3f kRVector = kIncr*m_spkCamera->GetRVector();
    m_spkCamera->SetAxes(kDVector,kUVector,kRVector);
}

void WindowApplication3::LookDown ()
{
    Matrix3f kIncr(m_akWorldAxis[2],m_fRotSpeed);

    Vector3f kDVector = kIncr*m_spkCamera->GetDVector();
    Vector3f kUVector = kIncr*m_spkCamera->GetUVector();
    Vector3f kRVector = kIncr*m_spkCamera->GetRVector();
    m_spkCamera->SetAxes(kDVector,kUVector,kRVector);
}

```

Notice that the incremental rotations are calculated about the world's right vector. The other two world axis directions must not change! The incremental rotation is designed to rotate only the camera coordinate frame.

I do not provide implementations for roll rotations about the world direction vector.

The member function MoveCamera ties the camera motion functions to key press events:

```

bool WindowApplication3::MoveCamera ()
{
    if ( !m_bCameraMoveable ) return false;
    bool bMoved = false;
    if ( m_bUArrowPressed ) { MoveForward(); bMoved = true; }
    if ( m_bDArrowPressed ) { MoveBackward(); bMoved = true; }
    if ( m_bHomePressed ) { MoveUp(); bMoved = true; }
    if ( m_bEndPressed ) { MoveDown(); bMoved = true; }
    if ( m_bLArrowPressed ) { TurnLeft(); bMoved = true; }
    if ( m_bRArrowPressed ) { TurnRight(); bMoved = true; }
    if ( m_bPgUpPressed ) { LookUp(); bMoved = true; }
    if ( m_bPgDnPressed ) { LookDown(); bMoved = true; }
    return bMoved;
}

```

The up and down arrow keys control forward and backward translation. The home and end keys control up and down translation. The left and right arrow keys control rotation about the up vector. The page up and page down keys control rotation about the right vector.

You will notice the use of the remaining eight Boolean data members: `m_bUArrowPressed`, `m_bDArrowPressed`, `m_bLArrowPressed`, `m_bRArrowPressed`, `m_bPgUpPressed`, `m_bPgDnPressed`, `m_bHomePressed`, and `m_bEndPressed`. These exist solely to avoid a classic problem in a real-time application when the operating system and event system are inherently not real time: The keyboard events are not processed in real time. If you implement `OnSpecialKeyDown` to include calls to the actual transformation function such as `MoveForward`, you will find that the camera motion is not smooth and appears to occur in spurts. The problem is the speed at which the windowing system processes the events and dispatches them to the event handler. The workaround is to use the `OnSpecialKeyDown` and `OnSpecialKeyUp` only to detect the state of the special keys: down or up, pressed or not pressed. The function call `MoveCamera` is made *inside the idle loop*. When the up arrow key is pressed, the variable `m_bUArrowPressed` is set to true. As long as the key is pressed, that variable is constantly set to true at the frequency the events are processed. However, from the idle loop's perspective, the value is a constant true. The `MoveForward` function is called at the frequency the idle loop is called at—a rate that is much larger than that of the event system. The result is that the camera motion is smooth. When the up arrow key is released, the variable `m_bUArrowPressed` is set to false, and the calls to `MoveCamera` in the idle loop no longer translate the camera.

The translation and rotation speeds are adjustable at run time. My default implementation of `OnKeyDown` is

```
bool WindowApplication3::OnKeyDown (unsigned char ucKey,
    int iX, int iY)
{
    if (WindowApplication3::OnKeyDown(ucKey,iX,iY))
    {
        return true;
    }

    // standard keys
    switch ( ucKey )
    {
    case 't': // slower camera translation
        if (m_bCameraMoveable)
        {
            m_fTrnSpeed /= m_fTrnSpeedFactor;
        }
        return true;
    case 'T': // faster camera translation
```

```

        if (m_bCameraMoveable)
        {
            m_fTrnSpeed *= m_fTrnSpeedFactor;
        }
        return true;
    case 'r': // slower camera rotation
        if (m_bCameraMoveable)
        {
            m_fRotSpeed /= m_fRotSpeedFactor;
        }
        return true;
    case 'R': // faster camera rotation
        if (m_bCameraMoveable)
        {
            m_fRotSpeed *= m_fRotSpeedFactor;
        }
        return true;
    case '?': // reset the timer
        ResetTime();
        return true;
    };

    return false;
}

```

The call to `WindowApplication::OnKeyDown` is to detect if the ESC key has been pressed, in which case the application will terminate. The camera translation speeds are controlled by keys t and T; the camera rotation speeds are controlled by keys r and R.

Object Motion

An object in the scene can be rotated using keyboard or mouse events. Usually, the object is the entire scene. To allow object motion, call the function `InitializeObjectMotion` and pass the object itself. The data member `m_spkMotionObject` points to that object. The Boolean data member `m_bObjectMoveable`, whose default value is false, is set to true by the function call.

We must decide first what the semantics of the rotation are. If the object has a parent (i.e., it is not the root of the scene), then the coordinate system of the object is that of its parent. The columns of the parent's world rotation matrix are the coordinate axis directions. To be consistent with the choice made for classes `Camera` and `Light`, whenever a rotation matrix represents coordinate axes, column 0 is the *direction* vector, column 1 is the *up* vector, and column 2 is the *right* vector. *Roll* is

rotation about the direction vector, *yaw* is rotation about the up vector, and *pitch* is rotation about the right vector. Let Q be the rotation matrix about one of these axes by a predetermined angle. If R is the object's local rotation matrix, then the update of the local rotation matrix is $R \leftarrow QR$. If the object is the root of the scene, the world rotation matrix is the identity matrix. Column 0 is (1, 0, 0) and is the direction vector, column 1 is (0, 1, 0) and is the up vector, and column 2 is (0, 0, 1) and is the right vector. The incremental rotation matrix Q is computed using these axes and the predetermined angle.

To rotate the object via keyboard events, a mechanism similar to camera motion is used. The data members `m_iDoRoll`, `m_iDoYaw`, and `m_iDoPitch` are state variables that keep track of the pressed states of various keys. Roll is controlled by the F1 and F2 keys. If neither key is pressed, the default state for `m_iDoRoll` is zero. If F1 is pressed, `m_iDoRoll` is set to -1 . If F2 is pressed, `m_iDoRoll` is set to $+1$. The signed values indicate the direction of rotation about the axis of rotation: -1 for clockwise rotation, 0 for no rotation, and $+1$ for counterclockwise rotation. Similarly, *yaw* is controlled by the F3 key (`m_iDoYaw` is set to -1) and the F4 key (`m_iDoYaw` is set to $+1$), and *pitch* is controlled by the F5 key (`m_iDoPitch` is set to -1) and the F6 key (`m_iDoPitch` is set to $+1$). The state of the keys is detected in `OnSpecialKeyDown` and `OnSpecialKeyUp`, just as was the case for camera motion via arrow keys and other special keys. The state-tracking mechanism guarantees that the rotations occur in the idle loop and are not limited by the slower event handler.

The function `MoveObject`, called in the `OnIdle` callback, is used to update the local rotation of the object whenever one of the six function keys is pressed. Its implementation is

```
bool WindowApplication3::MoveObject ()
{
    if (!m_bCameraMoveable || !m_spkMotionObject)
    {
        return false;
    }

    Spatial* pkParent = m_spkMotionObject->GetParent();
    Vector3f kAxis;
    Matrix3f kRot, kIncr;

    if (m_iDoRoll)
    {
        kRot = m_spkMotionObject->Local.GetRotate();
        fAngle = m_iDoRoll*m_fRotSpeed;
        if (pkParent)
        {
            kAxis = pkParent->World.GetRotate().GetColumn(0);
        }
        else
```

```

    {
        kAxis = Vector3f::UNIT_X;
    }
    kIncr.FromAxisAngle(kAxis, fAngle);
    m_spkMotionObject->Local.SetRotate(kIncr*kRot);
    return true;
}

// ... Similar blocks for yaw and pitch go here ...

return false;
}

```

The rotation axis is computed according to my earlier description. The rotation angle is either plus or minus the rotation speed parameter; the choice of sign depends on which function key was pressed.

The `WindowApplication3::OnKeyDown` implementation allows you to adjust the object rotation speed when the function keys are pressed. This is accomplished by the `r` key (decrease the rotation speed) and the `R` key (increase the rotation speed).

The application class has some data members and functions to support rotating an object by the mouse. The rotation is accomplished via a virtual trackball that is manipulated by the callbacks `OnMouseClicked` and `OnMotion`. In order to rotate, the virtual trackball must be enabled (`m_bUseTrackBall` is set to true), there must be a motion object (`m_spkMotionObject` is not null), and the left mouse button must generate the events (input `iButton` must be `MOUSE_LEFT_BUTTON`).

The virtual trackball is assumed to be a sphere in the world whose projection onto the screen is a circle. The circle center is $(W/2, H/2)$, where W is the width of the screen and H is the height of the screen. The circle radius is $r = \min\{W/2, H/2\}$. Figure 18.3 shows a typical projection.

The trackball uses a right-handed, normalized coordinate system whose origin is the center of the circle and whose axis directions are parallel to the screen coordinate axes: x right and y up. Scaling occurs so that in this coordinate system the square is defined by $|x| \leq 1$ and $|y| \leq 1$. The circle itself is defined by $x^2 + y^2 = 1$. The starting point of a mouse drag operation is shown in Figure 18.3 and is labeled (x_0, y_0) . The ending point of the mouse drag is denoted (x'_1, y'_1) . Any point that is outside the circle is projected onto the circle. The actual ending point used by the trackball is labeled (x_1, y_1) in the figure.

Imagine the starting and ending points being located on the sphere itself. Since the circle is $x^2 + y^2 = 1$, the sphere is $x^2 + y^2 + z^2 = 1$. The z -axis is defined to be into the screen, so the hemisphere on which you select points satisfies $z \leq 0$. The (x, y, z) coordinate system is shown in Figure 18.3. The points (x_i, y_i) are mapped to the sphere points

$$\mathbf{V}_i = (x_i, y_i, -\sqrt{1 - x_i^2 - y_i^2})$$

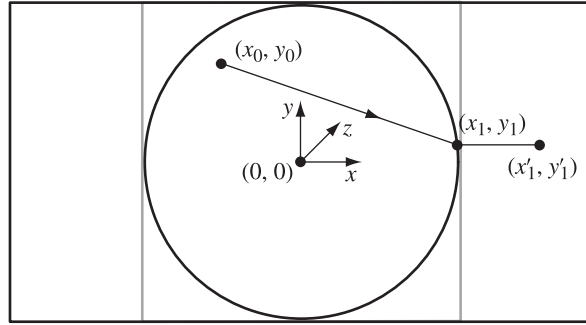


Figure 18.3 The projection of a virtual trackball onto the screen. The circle of projection is positioned at the center of the screen. The bounding square of the circle is shown.

The rotation of the trackball implied by the mouse drag has a rotation axis that is perpendicular to the plane containing the sphere center and the two sphere points. The rotation needs to be computed in world coordinates, so be aware that the cross product $\mathbf{V}_0 \times \mathbf{V}_1$ is in the normalized coordinate system, not in the world. A vector (x_i, y_i, z_i) in the normalized coordinate system corresponds to the world vector

$$\mathbf{W}_i = x_i \mathbf{R} + y_i \mathbf{U} + z_i \mathbf{D}$$

where $\mathbf{R}$, $\mathbf{U}$, and $\mathbf{D}$ are the camera's right, up, and direction vectors, respectively, in world coordinates. The cross product of the two vectors is

$$\begin{aligned} \mathbf{W}_0 \times \mathbf{W}_1 &= (x_0 \mathbf{R} + y_0 \mathbf{U} + z_0 \mathbf{D}) \times (x_1 \mathbf{R} + y_1 \mathbf{U} + z_1 \mathbf{D}) \\ &= (y_1 z_0 - y_0 z_1) \mathbf{R} + (x_0 z_1 - x_1 z_0) \mathbf{U} + (x_1 y_0 - x_0 y_1) \mathbf{D} \end{aligned}$$

The world coordinates $(y_1 z_0 - y_0 z_1, x_0 z_1 - x_1 z_0, x_1 y_0 - x_0 y_1)$ are *not* generated by $(x_0, y_0, z_0) \times (x_1, y_1, z_1)$. They are generated by $(z_0, y_0, x_0) \times (z_1, y_1, x_1)$. This has to do with the camera axis ordering $(\mathbf{D}, \mathbf{U}, \mathbf{R})$, which corresponds to a tuple (z, y, x) .

The rotation axis in world coordinates is $\mathbf{W}_0 \times \mathbf{W}_1$. The angle of rotation is the angle between the two vectors, $\theta = \cos^{-1}(\mathbf{W}_0 \cdot \mathbf{W}_1)$. The member function `RotateTrackBall` computes the axis, angle, and the corresponding rotation matrix; call it $\mathbf{Q}$. If the object is the root of the scene, the local rotation matrix $\mathbf{R}$ is updated just as we did for the keyboard-driven rotation: $\mathbf{R} \leftarrow \mathbf{Q}\mathbf{R}$.

If the object has a parent, the update is more complicated. The object has a model-to-world transformation that positions and orients the object in the world. If $\mathbf{X}$ is a model-space point, the corresponding world-space point is

$$\mathbf{Y} = \mathbf{R}_w \mathbf{S} \mathbf{X} + \mathbf{T}_w$$

where R_w is the world rotation, S is the diagonal scaling matrix, and T_w is the world translation. In all my sample applications, I arrange for the object's world translation to be the origin so that the rotation is about the origin. The process of doing so is sometimes referred to as a *center-and-fit* operation. After this operation, the world translation is $T_w = \mathbf{0}$, so

$$\mathbf{Y} = R_w S \mathbf{X}$$

The world rotation is constructed from the object's parent's world rotation and from its local rotation. The relationship is

$$R_w = R_p R_\ell$$

where R_p is the parent's world rotation and R_ℓ is the object's local rotation. The world point is therefore

$$\mathbf{Y} = R_p R_\ell S \mathbf{X}$$

The trackball motion applies a world-to-world rotation matrix, which we called Q . The transformation of the world point $\mathbf{Y}$ to the rotated point $\mathbf{Z}$ is

$$\mathbf{Z} = Q \mathbf{Y} = (Q R_p R_\ell) S \mathbf{X}$$

We do not want to modify the parent's rotation matrix by multiplying on the left by Q . Instead, we wish to adjust the local rotation R_ℓ while preserving the parent rotation. If R'_ℓ is the adjusted local rotation, we need

$$R_p R'_\ell = Q R_p R_\ell$$

Solving for the adjusted local rotation,

$$R'_\ell = (R_p^T Q R_p) R_\ell$$

The expression $R_p^T Q R_p$ is a *similarity transformation* and is viewed as the representation of the rotation Q in the coordinate system of the object. The matrix Q by itself represents the rotation in the parent's coordinate system.

The source code for `RotateTrackBall` is lengthy, but this is a straightforward implementation of the ideas discussed here:

```
void WindowApplication3::RotateTrackBall (float fX0, float fY0,
    float fX1, float fY1)
{
    if ((fX0 == fX1 && fY0 == fY1) || !m_spkCamera)
    {
        // nothing to rotate
        return;
    }
}
```

```

// Get first vector on sphere.
float fLength = Mathf::Sqrt(fX0 * fX0 + fY0 * fY0);
float fInvLength, fZ0, fZ1;
if (fLength > 1.0f)
{
    // Outside unit disk, project onto it.
    fInvLength = 1.0f/fLength;
    fX0 *= fInvLength;
    fY0 *= fInvLength;
    fZ0 = 0.0f;
}
else
{
    // Compute point (x0,y0,z0) on negative unit hemisphere.
    fZ0 = 1.0f - fX0 * fX0 - fY0 * fY0;
    fZ0 = (fZ0 <= 0.0f ? 0.0f : Mathf::Sqrt(fZ0));
}
fZ0 *= -1.0f;

// Use camera world coordinates; order is (D,U,R),
// so point is (z,y,x).
Vector3f kVec0(fZ0,fY0,fX0);

// Get second vector on sphere.
fLength = Mathf::Sqrt(fX1 * fX1 + fY1 * fY1);
if (fLength > 1.0f)
{
    // Outside unit disk, project onto it.
    fInvLength = 1.0f/fLength;
    fX1 *= fInvLength;
    fY1 *= fInvLength;
    fZ1 = 0.0f;
}
else
{
    // Compute point (x1,y1,z1) on negative unit hemisphere.
    fZ1 = 1.0f - fX1 * fX1 - fY1 * fY1;
    fZ1 = (fZ1 <= 0.0f ? 0.0f : Mathf::Sqrt(fZ1));
}
fZ1 *= -1.0f;

// Use camera world coordinates; order is (D,U,R),
// so point is (z,y,x).
Vector3f kVec1(fZ1,fY1,fX1);

```

```

// Create axis and angle for the rotation.
Vector3f kAxis = kVec0.Cross(kVec1);
float fDot = kVec0.Dot(kVec1);
float fAngle;
if (kAxis.Normalize() > Mathf::ZERO_TOLERANCE)
{
    fAngle = Mathf::ACos(kVec0.Dot(kVec1));
}
else // Vectors are parallel.
{
    if (fDot < 0.0f)
    {
        // rotated pi radians
        fInvLength = Mathf::InvSqrt(fX0 * fX0 + fY0 * fY0);
        kAxis.X() = fY0 * fInvLength;
        kAxis.Y() = -fX0 * fInvLength;
        kAxis.Z() = 0.0f;
        fAngle = Mathf::PI;
    }
    else
    {
        // rotation by zero radians
        kAxis = Vector3f::UNIT_X;
        fAngle = 0.0f;
    }
}

Vector3f kWorldAxis =
    kAxis.X() * m_spkCamera->GetDVector() +
    kAxis.Y() * m_spkCamera->GetUVector() +
    kAxis.Z() * m_spkCamera->GetRVector();

Matrix3f kTrackRotate(kWorldAxis, fAngle);

const Spatial* pkParent = m_spkMotionObject->GetParent();
Matrix3f kLocalRot;
if (pkParent)
{
    const Matrix3f& rkPRotate = pkParent->World.GetRotate();
    kLocalRot = rkPRotate.TransposeTimes(kTrackRotate) * rkPRotate *
        m_kSaveRotate;
}
else

```

```

    {
        kLocalRot = kTrackRotate * m_kSaveRotate);
    }
    kLocalRot.Orthonormalize();
    m_spkMotionObject->Local.SetRotate(kLocalRot);

    m_spkMotionObject->UpdateGS();
}

```

The first part of the code computes the world axis of rotation and the angle of rotation, just as I described. The last part of the code updates the object's local rotation based on the trackball motion. The incremental update matrix is Q if the object has no parent, but $R_p^T Q R_p$ if the object has a parent.

You should notice that the update uses a data member named `m_kSaveRotate`. The trackball rotation is always anchored to the starting point. When that point is selected, the object's current local rotation is saved. The trackball rotation is always applied to the original local rotation.

The mouse event callbacks are simple enough:

```

bool WindowApplication3::OnMouseClicked (int iButton, int iState,
    int iX, int iY, unsigned int)
{
    if (!m_bUseTrackBall
        || iButton != MOUSE_LEFT_BUTTON
        || !m_spkMotionObject)
    {
        return false;
    }

    float fMult =
        1.0f/(m_iWidth >= m_iHeight ? m_iHeight : m_iWidth);

    if (iState == MOUSE_DOWN)
    {
        // Get the starting point.
        m_bTrackBallDown = true;
        m_kSaveRotate = m_spkMotionObject->Local.GetRotate();
        m_fXTrack0 = (2 * iX - m_iWidth) * fMult;
        m_fYTrack0 = (2 * (m_iHeight - 1 - iY) - m_iHeight) * fMult;
    }
    else

```

```

    {
        m_bTrackBallDown = false;
    }

    return true;
}

bool WindowApplication3::OnMotion (int iButton, int iX, int iY)
{
    if (!m_bUseTrackBall
        || iButton != MOUSE_LEFT_BUTTON
        || !m_bTrackBallDown
        || !m_spkMotionObject)
    {
        return false;
    }

    // Get the ending point.
    float fMult =
        1.0f/(m_iWidth >= m_iHeight ? m_iHeight : m_iWidth);
    m_fXTrack1 = (2 * iX - m_iWidth) * fMult;
    m_fYTrack1 = (2 * (m_iHeight - 1 - iY) - m_iHeight) * fMult;

    // Update the object's local rotation.
    RotateTrackBall(m_fXTrack0, m_fYTrack0, m_fXTrack1, m_fYTrack1);
    return true;
}

```

When the left mouse button is pressed, the callback `OnMouseClicked` is executed. The trackball is set to its down state, the object's local rotation R is saved, and the starting point is calculated from the screen coordinates of the mouse click. As the mouse is dragged, the callback `OnMotion` is continually called. It computes the ending point, and then calls the trackball rotation function to compute the incremental rotation Q and update the object's local rotation to QR .

Performance Measurements

Nearly all my sample applications include measuring the frame rate using function calls in the `OnIdle` callback. The rate is stored in the data member `m_dFrameRate`. The data member `m_dAccumulatedTime` stores the accumulated time used by the application. The function `MeasureTime` updates the accumulated time with the time difference between the last time the function was called, a value stored in `m_dLastTime`, and

the current time as read by `System::GetTime`. The data member `m_iFrameCount` stores the number of times `OnIdle` has been called and is the number of frames displayed. The frame rate is the ratio of the number of the frame count and the accumulated time.

Because the application can run quite rapidly, the resolution of the `System::GetTime` call might be too coarse to use on a frame-by-frame basis. If you have concerns about the resolution, I provided a miniature timer that is designed to have the accumulated time updated less frequently than each frame. The data member `m_iTimer` is the clock counter. The starting value for the counter is stored in the data member `m_iMaxTimer`. My libraries use a default of 30, but you are certainly welcome to set it to a different value in the constructor of your application. The idea is that `m_iTimer` is decremented on each frame. Once it reaches zero, `System::Time` is read, the accumulated time is updated, and `m_iTimer` is reset to `m_iMaxTimer`. The `MeasureTime` implementation is

```
void WindowApplication3::MeasureTime ()
{
    // Start performance measurements.
    if (m_dLastTime == -1.0)
    {
        m_dLastTime = System::GetTime();
        m_dAccumulatedTime = 0.0;
        m_dFrameRate = 0.0;
        m_iFrameCount = 0;
        m_iTimer = m_iMaxTimer;
    }

    // Accumulate the time only when the miniature timer allows it.
    if (--m_iTimer == 0)
    {
        double dCurrentTime = System::GetTime();
        double dDelta = dCurrentTime - m_dLastTime;
        m_dLastTime = dCurrentTime;
        m_dAccumulatedTime += dDelta;
        m_iTimer = m_iMaxTimer;
    }
}
```

The initial value of `m_dLastTime` is `-1`, indicating that the frame rate measuring system is uninitialized. You may reset the measuring system via `ResetTimer`, which just sets the last time to the invalid `-1`. A convenience function for displaying the frame rate in the lower-left corner of the screen is `DrawFrameRate`.

A convenience key is provided for the `ResetTimer` call, namely, the question mark (?) key.

18.9.6 MANAGING THE ENGINES

The two parts of the application layer you care about the most are the idle loop and the event handlers. These are both part of the message pump mentioned previously.

```
do_forever
{
    if ( message pending )
    {
        (6) If message is to quit, break out of loop;
        (7) Dispatch the message;
    }
    else
    {
        (8) Do idle processing;
    }
}
```

Events such as keystrokes and mouse clicks will be detected by the operating system and sent to the relevant applications through the message dispatch system. Callbacks will be executed such as `OnKeyPress` or `OnMousePress`. An operating system might have native support for gamepads and joysticks but, if not, there is always some mechanism for polling the hardware to check its status. The polling code can be added to the application layer, or it can be part of a callback that is executed during idle time. In Wild Magic's application layer, this callback is `OnIdle`. It is also possible that the input devices are managed by another API. For example, in addition to Direct3D graphics, you have `DirectInput` for accessing input devices.

The OnIdle Callback

For a single-threaded application, the callback `OnIdle` is the place where you manage all the subsystems for the application. All the Wild Magic graphics samples have code blocks of the form

```
void DefaultShader::OnIdle ()
{
    MeasureTime();

    if (MoveCamera())
    {
        m_kCuller.ComputeVisibleSet(m_spkScene);
    }
}
```



```

    if (MoveObject())
    {
        m_spkScene->UpdateGS();
        m_kCuller.ComputeVisibleSet(m_spkScene);
    }

    m_pkRenderer->ClearBuffers();
    if (m_pkRenderer->BeginScene())
    {
        m_pkRenderer->DrawScene(m_kCuller.GetVisibleSet());
        DrawFrameRate(8, GetHeight()-8, ColorRGBA::WHITE);
        m_pkRenderer->EndScene();
    }
    m_pkRenderer->DisplayBackBuffer();

    UpdateFrameCount();
}

```

The first and last functions are used to coarsely measure the frame rate and the function `DrawFrameRate` just displays the result in the lower-left corner of the screen.

The function `MoveCamera` checks to see if the camera has moved as a result of transformation changes that occurred because of keystrokes (arrow keys, page up or page down, home or end). If the camera has moved, the potentially visible set needs updating by a call to `ComputeVisibleSet`.

The function `MoveScene` checks to see if the scene has moved as a result of transformation changes that occurred because of mouse drags (via the virtual trackball) or by keystrokes (the first six function keys). If any transformation in the scene has changed, the scene graph is told to update its geometric state. The potentially visible set needs updating by a call to `ComputeVisibleSet`. Scene graph motion by the trackball or keystrokes causes local transformations to be set, but the call to the geometric update function `UpdateGS` is deferred until the idle loop. This is a design choice for my application layer. The idea is that you can set multiple local transformations and call `UpdateGS` once rather than calling `UpdateGS` after each local transformation is set. However, the setting of local transformations within your own application might have patterns that allow you to implement smart updates, calling `UpdateGS` only when needed and only at a minimum set of nodes in the scene.

Managing the Graphics Engine

The remaining block of code in `OnIdle` to discuss is the drawing pass. The function `ClearBuffers` clears the color, depth, and stencil buffers. Once rendering to the back buffer is completed, the function `DisplayBackBuffer` informs the graphics system that it is time to transfer the back buffer to the front buffer for display. Both of these

functions are implemented by the derived-class renderers. The pair of functions `BeginScene` and `EndScene` are implemented only by the `Direct3D` renderer to support similarly named function calls in the `Direct3D` API. The `OpenGL` and software renderers do nothing during these calls. All rendering function calls must occur between this pair of functions. In the current example, the potentially visible set is passed to the renderer. The objects in this set are drawn by the renderer, adhering to any semantics implied by the existence of global effects.

Managing the Physics Engine

Many of the physics sample applications on the CD-ROM have an `OnIdle` function of the following form:

```
void SomePhysicsApplication::OnIdle ()
{
    MeasureTime();
    DoPhysical();
    DoVisual();
    UpdateFrameCount();
}
```

The function `DoVisual` is a wrapper for the graphics portion of the processing.

```
m_pkRenderer->ClearBuffers();
if (m_pkRenderer->BeginScene())
{
    m_pkRenderer->DrawScene(m_kCuller.GetVisibleSet());
    DrawFrameRate(8,GetHeight() - 8,ColorRGBA::WHITE);
    m_pkRenderer->EndScene();
}
m_pkRenderer->DisplayBackBuffer();
```

The function `DoPhysical` encapsulates three things:

1. Simulating the physics.
2. Processing the results of the simulation to update the geometric state of the scene graphs.
3. Updating the potentially visible set.

How the scene graphs change depends on the physical simulation, so it makes sense that `DoPhysical` encapsulates this in a way that minimizes the work of updating. In my physics samples, the physical simulations themselves are nearly always separate components that are neatly encapsulated to deal only with the physics. The separation

between the graphics (visual display) and the physics (simulation) is important. The physics components tend to occur in files named `PhysicsModule`.

In a large-scale game application, the physical simulations can be quite complicated, so a simple factorization, as the previous pseudocode showed, is not necessarily possible. However, you should try as much as possible to maintain a separation of concerns and not intermingle graphics and physics code.

Managing Other Engines

The state changes to the scene graphs, whether geometric state or renderer state, can be driven by the event-handling system (keystrokes, mouse clicks, gamepad events). In a game application, there are other sources for state changes. For example, you might have an indoor level with a collection of rooms and hallways. The good guy (your character, of course) walks down a hall and turns into a room. A bad guy comes out of the shadows of the room to attack you. This happened because your walking through the doorway to the room *triggered* the event. A game can have a lot of triggers in it. The logic for managing these is specific to your game and, perhaps, considered part of your *artificial intelligence* (AI) engine. Naturally, there is a lot more to AI engines, but that is not my expertise. You will need to read other books for information on building such engines. That said, integration into a Wild Magic graphics application should not be difficult.

You might also have 3D sound capabilities in your application, the system encapsulated in a *sound engine*. Wild Magic is actually designed to support this. The `Spatial` class is the base class for internal nodes, `Node`, and for leaf nodes representing drawable data, `Geometry`. The intent of the scene graph core classes is to also allow a class `SoundEmitter`, which is derived from `Spatial`. The `Camera` class identifies the region of space in which you can see things. The `Geometry` objects in this region are drawn by the *graphics renderer*. The analogous class in the sound system is `Listener`. It defines the region of space in which you can hear things. The `SoundEmitter` objects in this region are played by the *sound renderer*. The NetImmerse engine had such a system, built on top of Intel's RSX sound system, which used an ellipsoidal model for the region in which you can hear things. Later, we supported 3D sound hardware from Aureal, but that company closed its doors. The sound system in NetImmerse finally used the Miles Sound System from Rad Games Tools. Gamebryo, the successor to NetImmerse, still uses this. A sound system in Wild Magic may be built using an abstract sound renderer API with back ends implemented for each platform of interest. On a Windows PC, you can use DirectSound to implement the back end. Other possibilities are to build the back end with OpenAL or with FMOD.

Yet one more engine to integrate would be a *networking engine*, whether it be for an application supporting a few machines or for a client-server-based application for a massively multiplayer online game. This also is not my area of expertise, but the clean design of the application layer makes it easy to integrate a networking engine into Wild Magic applications.

Asynchronous Behavior

Although `OnIdle` gives you the impression of a list of programming instructions that must be executed in sequential order, the order of execution is not necessarily that. When `MoveCamera` is called, the keyboard is not polled directly. The keystrokes that are tied into camera motion occur as hardware-generated interrupts, which are then mapped to messages in the event-handling system. The key press messages are dispatched by the message pump. The event handler receives these and calls the appropriate Wild Magic callback functions `OnKeyDown`, `OnKeyUp`, `OnSpecialKeyDown`, or `OnSpecialKeyUp`. For example, suppose that the up arrow key is pressed, indicating a desire to move the camera forward. This key is processed by `OnSpecialKeyUp` and `OnSpecialKeyDown`. The default implementation of `OnSpecialKeyUp` is in the class `WindowApplication3`, which sets a Boolean flag `m_bUArrowPressed` to true when the up arrow key has been pressed. The Boolean flag remains true until the key is released, a hardware interrupt is generated, a message corresponding to this is created and dispatched by the message pump, the event handler has received the message, and the callback `OnSpecialKeyDown` has executed. This callback sets `m_bUArrowPressed` to false, indicating the up arrow key is no longer depressed. The key press/release events occur much less frequently than calls to `OnIdle`. While the up arrow is depressed, `m_bUArrowPressed` remains true. The call to `MoveCamera` in `OnIdle` tests to see if `m_bUArrowPressed` is true, in which case the application callback `MoveForward` is executed. Thus, `MoveCamera` polls the keyboard *indirectly* in the sense of testing an application-supplied Boolean variable whose value represents some keyboard state.

You might have asked the question, Why not call `MoveCamera` in the key-event callback `OnSpecialKeyDown`? Try it yourself and see what the results are! What you will discover is that the camera motion is choppy. Effectively, the event handler and the idle loop are tasks that run asynchronously. The event handler gets called at a much less frequent rate than `OnIdle`. By calling `MoveCamera` in the key-event callback, the camera is moved infrequently compared to the rate at which `OnIdle` moves the camera. In this scenario, the call `MoveCamera` effectively does poll the keyboard, being called only at the rate the key events are occurring. This throws a wrench into your attempt to run at real-time rates with smooth camera motions.

This example shows that asynchronous tasks must communicate in a manner that does not (significantly) slow down the application. The application data member `m_bUArrowPressed` provides a means to communicate. Think of it as a mailbox. The event handler is the postal delivery person, depositing mail in your mailbox once a day—on a schedule unknown to most people. You may check the mailbox whenever you have the time and as often as you wish. This allows you to focus on other equally (or more) important tasks. If you so desired, you could ride around the city in the postal delivery person's truck. When he stopped at your mailbox, you would know exactly the right time to check for new mail. However, riding around in a truck is not an effective use of your time.

The `OnIdle` discussion so far has been in the context of an application running in a single thread. Despite this, there is asynchronicity inherent in the system. This

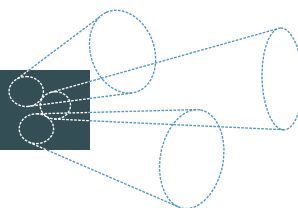
becomes explicit when you decide to write applications that launch other threads. Each thread performs some task (or tasks), the results of which are interesting to the application itself. Rather than have the application poll the threads (if that is actually possible) to see if their results are ready to use, a communication mechanism much like that used for key press/release events is useful. This mechanism might even be as simple as providing a Boolean flag such as `m_bPhysicsResultsAvailable`. The flag is set to `false` when the thread is launched to do its thing. The application polls the flag in the `OnIdle` function. Once the flag is detected as `true`, the application responds accordingly, whether this be calling functions within its own thread or launching new threads. The thread whose results just became available most likely will have associated with it other data members in which to store its results. These, of course, will be accessible to the application. In this sense, the Boolean flag also acts as a “dirty bit,” indicating whether or not the thread’s data is current.

The application itself might be given a way to communicate with the other subsystems. For example, the `DisplayBackBuffer` call that is made after a scene is rendered is a *request* to the graphics subsystem to transfer the contents of the back buffer to the front buffer. Nothing guarantees this will happen immediately. The request can be queued up by the graphics subsystem for later processing when the subsystem has the time.

How you structure your application, how you use the various engines, and how you manage threads and processes is highly dependent on what your application does. The `OnIdle` loop is where all the logic resides, albeit via high-level function calls to the various engines.

CHAPTER

19



MEMORY MANAGEMENT

The majority of the discussion in this book about resource management has to do with data that is transferred to video memory to be used by the graphics system. This data includes vertex and index buffers, vertex and shader programs, and texture images. Memory management in video memory is the responsibility of the graphics drivers. Whether the data resides in video memory or in accelerated graphics port (AGP) memory is up to the drivers to decide, although you might be allowed to provide hints about where to store items based on how frequently you plan on modifying those items. On the other hand, system memory is your responsibility. This is yet another resource that must be properly managed. This chapter contains some basic information about memory management. The topic is quite broad and has been studied extensively over the years. For a broader and deeper discussion than that provided here, you will need to do some research on your own. As always, the quintessential reference is Donald Knuth's famous book series [Knu73]. (For online references, using your favorite Internet search engine, a search on the key words "memory management" will produce a significant number of hits. A well-written online reference is [Lim01].)

19.1 MEMORY BUDGETS FOR GAME CONSOLES

When you are raised on desktop computers, memory management is a simple matter of allocating and deallocating memory from the global heap whenever you must. In C, you do this with `malloc` and `free`. In C++, you do this with calls to the global `new` and `delete` operators. Desktop computers tend to have an enormous amount of physical memory, so you probably are infrequently concerned about exceeding the limits.

I had not thought much about memory usage until working on visualization and analysis of medical image data sets. The 3D images are quite large. Computational geometry algorithms applied to the data sets usually require adjacency information for surface meshes extracted from the images. The meshes can have on the order of millions of vertices, edges, and triangles. These all use significant amounts of memory, so much that my main reason for upgrading computer motherboards was to get more memory slots and support larger amounts of memory (my current medical image analysis development machine has 4 gigabytes of memory). Larger and faster throughput of triangles and larger amounts of video memory on graphics cards also cause me to upgrade regularly, whether for computer graphics and games contract programming or for medical image analysis (my development machines are now running on PCI Express rather than AGP).

Developing on machines with a large amount of memory (and fast CPUs) is a double-edged sword. On one hand, you have enough memory not to worry much about using it all. On the other hand, if you create applications that run on lower-end machines with a smaller amount of memory, careless disregard about memory usage might cause such applications to crash because its memory requirements just cannot be met. Naturally, your applications should go to great pains not to crash in such situations.

Game consoles are different beasts than desktop computers. Invariably, the consoles have much more processing power, usually because they have multiple processors, and they have much less physical memory. The mind-set you have when developing on a machine with 64MB of memory must be different from that when developing on a machine with 4GB of memory.

During development of a console game, it is quite reasonable to specify a *memory budget* for each system/engine used in the game. The graphics engine, physics engine, and sound, AI, networking, and other systems need to stay within their budgets. On a first game, it is not clear how you determine the budgets. For companies that develop game sequels, you at least have the previous version of the game to give you an idea of how to partition the console memory into chunks for each of the systems. How you partition memory is not a topic I will cover.

As I said previously, desktop development tends to use memory allocation and deallocation from a global heap. If you were told that you had a budget of N megabytes, how would you guarantee that you stayed within your budget? If you were to exceed your budget, what would you do? What you need to do is encapsulate the memory management into a system that supports budgets.

One way to accomplish this is to create a wrapper around the global heap management, say, by a class `MemoryManager`. Instead of calling the global `new` and `delete` operators yourself, `MemoryManager` provides member functions for allocation and deallocation, each function calling `new` and `delete` as needed but also tracking the requests. These functions force you to specify the system that needs the memory (graphics, physics, sound, etc.). The class logs information about the requests and keeps track of how much memory each system is currently using. If a request for allocation exceeds the system's budget, the allocation fails, even though the global heap might have sufficient memory to support the request. The bottom line is that you cannot

borrow memory from other systems. You get what you get. During development, if your system often attempts to exceed its budget, you most likely need to sit down with the game designers and explain why your budget needs to be revised. Just like in any other commercial application, the shipped product needs to be robust. If an allocation fails, the application must handle this gracefully. How you do that depends on your application and what the program is trying to do at the time.

The advantage to a class like `MemoryManager` is that you can rely on the operating system and run-time libraries to handle the details of memory allocation and deallocation. The disadvantage is that you have the potential for fragmentation of a system's memory. For example, if you have systems A, B, and C, and each in turn requests a block of memory, the global heap has used blocks organized as

$$A_0, B_0, C_0, A_1, B_1, C_1, A_2, B_2, C_2, \dots$$

where the A_i are memory blocks for system A; B_i are memory blocks for system B; and C_i are memory blocks for system C. This type of fragmentation is not the same as fragmentation of the global heap. In the aforementioned example, all the blocks fill a contiguous portion of the global heap, so there is no heap fragmentation. But the blocks for a single system are scattered about the heap. This has the potential for generating cache misses during program execution.

The alternative is to allocate *local heaps* for the systems. In its simplest form, each system receives a single contiguous block of memory (a local heap) from the global heap. The system may use only that memory. This can help to reduce fragmentation within a system. Moreover, it allows you to easily shut down (or restart) a system—deallocation of all the subblocks is simply not required. Instead, the local heap manager can reset its data structures. Of course, this alternative requires you to write a memory manager for the local heap.

C++ has the concept of implementing the `new` and `delete` operators within a class. The idea is that the class knows about its memory patterns and usage, something the global operators do not. The class may take advantage of this, essentially providing its own local heap (allocated from the global heap) and memory management. Although this is a useful concept, the problem in a system such as the graphics system is that the collection of objects to be allocated and deallocated is heterogeneous. A single memory manager will not suffice for the system.

19.2 LEAK DETECTION AND COLLECTING STATISTICS

Before talking about the general concepts involved in building a memory management system, let us take a closer look at the C++ mechanism for allocating and deallocating memory via the global `new` and `delete` operators. The C++ language allows you to create allocation functions referred to as *placement new* operators. You may implement these to provide additional information when allocating memory and to include your own flavor of memory tracking and management. Wild Magic

implements these to check for memory leaks and to collect statistics about memory usage. The statistical information is not extensive, but the source code is easy to modify, so you can include as much gathering of information as you like—even to the point of logging each and every memory allocation and deallocation in order to analyze the access patterns and memory utilization.

The Wild Magic memory manager is encapsulated in the class named `Memory`, found in the Foundation library `System` folder. The class represents a singleton—all data members are static. By default, the memory manager is disabled. The macros `WM4_NEW` and `WM4_DELETE` expand to `new` and `delete`, respectively. To enable the system, you must define the preprocessor symbol `WM4_MEMORY_MANAGER`. I do so by including two build configurations, `Debug Memory` and `Release Memory`, in the core libraries. The configurations define the preprocessor symbol in the project properties. The applications have build configurations that are dependent on the graphics API, so you will see configurations such as `Dx9 Debug Memory` and `Wgl Release Memory`. The Linux makefiles and the Macintosh Xcode projects have similarly named configurations.

When the memory manager is enabled, the macros are defined as shown next. Moreover, the global `new` and `delete` operators are overridden and placement `new` and `delete` operators are defined.

```
#define WM4_NEW new(__FILE__,__LINE__)
#define WM4_DELETE delete
void* operator new (size_t uiSize);
void* operator new[] (size_t uiSize);
void* operator new (size_t uiSize, char* acFile, unsigned int uiLine);
void* operator new[] (size_t uiSize, char* acFile, unsigned int uiLine);
void operator delete (void* pvAddr);
void operator delete[] (void* pvAddr);
void operator delete (void* pvAddr, char* acFile, unsigned int uiLine);
void operator delete[] (void* pvAddr, char* acFile, unsigned int uiLine);
```

An allocation using `WM4_NEW` leads to a call of the third or fourth `new` operator in the list. The macro `__FILE__` expands to the file name containing the line of code that has the allocation. The macro `__LINE__` expands to the line number for that line of code. The `uiSize` parameter is automatically generated by the compiler since it knows how large a memory chunk is needed to store a to-be-allocated object or an array of to-be-allocated objects. The `acFile` parameter receives the file name and the `uiLine` parameter receives the line number.

As an example, suppose that the placement `new` operator is called (as compared to the placement `new[]` operator). The implementation of this function is

```
void* operator new (size_t uiSize, char* acFile, unsigned int uiLine)
{
    return Memory::Allocate(uiSize,acFile,uiLine,false);
}
```

All the semantics of memory allocation and of the gathering of statistical information is wrapped up in the Memory function `Allocate`. The last parameter is a Boolean whose value is false if `new` is called but true if `new[]` is called. I track this to see if mismatches have occurred, pairing a call to `new` with a call to `delete[]` or pairing a call to `new[]` with a call to `delete`. Use your imagination to come up with all sorts of Memory-like classes you can build to track allocations and deallocations.

My own allocation wrapper is

```
void* Memory::Allocate (size_t uiSize, char* acFile,
    unsigned int uiLine, bool bIsArray)
{
    ms_uiNumNewCalls++;

    // The 'assert' used to be enabled to trap attempts to
    // allocate zero bytes. However, the DirectX function
    // D3DXCheckNewDelete may pass in a value of zero.
    // assert(uiSize > 0);

    // Allocate additional storage for the block header
    // information.
    size_t uiExtendedSize = sizeof(Block) + uiSize;
    char* pcAddr = (char*)malloc(uiExtendedSize);

    // Save the allocation information.
    Block* pkBlock = (Block*)pcAddr;
    pkBlock->Size = uiSize;
    pkBlock->File = acFile;
    pkBlock->Line = uiLine;
    pkBlock->IsArray = bIsArray;
    InsertBlock(pkBlock);

    // Move the pointer to the start of what the user expects
    // from 'new'.
    pcAddr += sizeof(Block);

    // Keep track of the number of allocated blocks and bytes.
    ms_uiNumBlocks++;
    ms_uiNumBytes += uiSize;

    if (ms_uiMaxAllowedBytes > 0
        && ms_uiNumBytes > ms_uiMaxAllowedBytes)
    {
        // The allocation has exceeded the maximum number of
        // bytes.
```

```

        assert(false);
    }

    // Keep track of the maximum number of bytes allocated.
    if (ms_uiNumBytes > ms_uiMaxAllocatedBytes)
    {
        ms_uiMaxAllocatedBytes = ms_uiNumBytes;
    }

    // Keep track of the distribution of sizes for allocations.
    if (ms_bTrackSizes)
    {
        // Keep track of the largest block ever allocated.
        if (uiSize > ms_uiMaxBlockSize)
        {
            ms_uiMaxBlockSize = uiSize;
        }

        unsigned int uiTwoPowerI = 1;
        int i;
        for (i = 0; i <= 30; i++, uiTwoPowerI <= 1)
        {
            if (uiSize <= uiTwoPowerI)
            {
                ms_auiHistogram[i]++;
                break;
            }
        }
        if (i == 31)
        {
            ms_auiHistogram[i]++;
        }
    }

    return (void*)pcAddr;
}

```

The function has a mixture of memory management and statistical gathering. Let's look at the memory management first. Abstractly, a call to the global `new` operator resolves to a request for a specific number of bytes from memory. A run-time library might very well use `malloc` to allocate the memory and just return the pointer to the memory block. My manager actually allocates the requested number of bytes, *and it prepends extra bytes that store information about the request*. The extra bytes are structured according to the nested struct `Memory::Block`,

```

class Memory
{
public:
    struct Block
    {
        size_t Size;
        const char* File;
        unsigned int Line;
        bool IsArray;
        Block* Prev;
        Block* Next;
    };
};

```

In `Memory::Allocate`, the size of the requested memory plus the size for a `Block` is stored in `uiExtendedSize`. The allocation occurs using `malloc`; do not call `new` here because you will get an infinitely recursive call! The allocated memory is typecast to a `Block` to allow you to store statistical information in that block. The struct member `Size` stores the requested number of bytes for the allocation. The member `File` stores a pointer to the file name. The file name is stored as global data, so `File` is not a dynamically allocated field of `Block`. The member `Line` stores the line number of the file where the allocation occurs. The member `IsArray` indicates whether `new` or `new[]` was called.

The last two data members of `Block`—namely, `Prev` and `Next`—support keeping track of the memory blocks using a doubly linked list. This is *not* an attempt to usurp the responsibilities of the global heap manager. The doubly linked list exists only to efficiently generate a report at the end of the application. It is expected that the list of blocks is empty. If it is not, a traversal of the list is performed and the block information is written to disk. The information includes the file names and line numbers where the blocks were allocated, hopefully giving you some valuable information that lets you discover why the blocks were not deallocated. The report is generated at any time by a call to `Memory::GenerateReport`.

After the `Block` information is initialized, it is necessary to return a pointer to the caller that points to the actual data it expects—the memory occurring immediately after the `Block` header. This happens by incrementing the full block pointer `pcAddr` by the size of the header.

The deallocation that is paired with the allocation mentioned previously is

```

void operator delete (void* pvAddr)
{
    Memory::Deallocate((char*)pvAddr, false);
}

```

The semantics of deallocation and statistical information gathering are also handled by the `Memory` class. The deallocation function is

```
void Memory::Deallocate (char* pcAddr, bool bIsArray)
{
    ms_uiNumDeleteCalls++;

    if (!pcAddr)
    {
        return;
    }

    // Move the pointer to the start of the actual allocated block.
    pcAddr -= sizeof(Block);

    // Get the allocation information and remove the block. The removal
    // modifies only the Prev and Next pointers, so the block information is
    // accessible after the call.
    Block* pkBlock = (Block*)pcAddr;
    RemoveBlock(pkBlock);

    // Check for correct pairing of new/delete or new[]/delete[].
    assert(pkBlock->IsArray == bIsArray);

    // Keep track of the number of allocated blocks and bytes. If the number
    // of blocks is zero at this time, a delete has been called twice on the
    // same pointer. If the number of bytes is too small at this time, some
    // internal problem has occurred within this class and needs to be
    // diagnosed.
    assert(ms_uiNumBlocks > 0 && ms_uiNumBytes >= pkBlock->Size);
    ms_uiNumBlocks--;
    ms_uiNumBytes -= pkBlock->Size;

    // Deallocate the memory block.
    free(pcAddr);
}
```

Regarding the memory management itself, the `Memory::Allocate` function called `malloc` to allocate a block of the requested size, including enough memory to store a `Block` header. The `pcAddr` pointer was incremented to point to the memory that the application expects. Before the memory can be deallocated, `pcAddr` must be decremented to point to the beginning of the header block. Once all statistical information is computed, a call to `free` causes the memory to be deallocated.

The statistics collected by the memory manager include counting how many times `new` or `new[]` is called (`ms_uiNumNewCalls`) and how many times `delete` or `delete[]` is called (`ms_uiNumDeleteCalls`). The number of currently active blocks is maintained (`ms_uiNumBlocks`), as well as the number of currently used bytes (`ms_uiNumBytes`). The number of bytes includes what the user requested but not the amount of memory used by the Block headers.

I also allow the user to specify the maximum amount of memory used by the entire engine by setting `ms_uiMaxAllocatedBytes`. When exceeded, an assert is triggered in debug mode. In an application that must not exceed its budget, you probably will need to modify the Memory class to gracefully exit (no memory allocated) and the application must respond to the crisis accordingly.

Finally, I keep track of the maximum-size block ever allocated (`ms_uiMaxBlockSize`). I also keep track of the number of allocated blocks of various sizes. The array element `ms_auiHistogram[0]` stores the number of allocated blocks of size 1. The array element `ms_auiHistogram[31]` stores the number of allocated blocks of size larger than 2^{30} . For $1 \leq i \leq 30$, the element `ms_auiHistogram[i]` stores the number of allocated blocks of size n , where $2^{i-1} < n \leq 2^i$. If you discover that you have a large number of allocations of small chunks of memory, you will want to understand why this is occurring and formulate a plan on how to avoid this.

The information I store is relatively minimal. As you increase the amount of information to be stored, the block header size must increase. This measurement system itself affects memory usage, so you need to be careful about the memory usage by the memory manager on a machine with a small amount of memory.

Some memory management systems make an attempt to detect *memory overruns*. You can have a Header class to store information about the allocated memory block, but the end of the structure has a fixed number of bytes, called *guard bytes*, that are initialized to some known values. A Header object occurs before the user-requested memory block. You can also have a Footer class to store information after the user-requested memory block. This might be something as simple as a fixed number of guard bytes also initialized to some known values. When a memory block is deallocated, a check is made to see if the guard bytes have changed values. If they have, then memory was written to it that should not have been. When running in debug mode in Microsoft Visual Studio, you may have seen messages about heap corruption. The memory manager is set up to use guard bytes and in this case has detected that a memory overrun has occurred.

A system that uses guard bytes is not perfect. If your application incorrectly writes to memory in the Header block *before* the guard bytes, the error is not detected on deallocation, but the header information has been destroyed. In my Memory class, if the block size is accidentally overwritten, the deallocation is incorrect, most likely resulting in some type of crash. You can be more sophisticated in the design of header and footer information, but at the cost of performance during debug runs. For example, you could use a *cyclic redundancy check* (CRC) by storing derived information from the header/footer on allocation and then verifying the derived information has not changed on deallocation. The CRC information most likely is not stored with the

allocated block, but in some location that (hopefully) is far enough away from the actively used memory to protect it from corruption itself.

If you want fine-tuned information about memory usage, you could also have a logging system that writes to disk information about each and every allocation/deallocation call. This is very slow if you write to disk every time, but you can attempt to have a *disk cache*, either provided by the hardware itself or by a virtual disk in system memory. You amortize the costs by writing to the cache, flushing it to disk when full. Statistical analyses may be applied to the logged information to give you an idea of how your game application is using (or abusing) memory.

19.3 GENERAL MEMORY MANAGEMENT CONCEPTS

The issues in building a memory management system are the same as those for building other software components. You have a number of goals to achieve and not all of them are possible simultaneously. There are trade-offs to consider. Your choices need to be based on understanding how such a system will be used in your applications.

The three main issues to deal with are *memory utilization*, *speed of allocation/deallocation*, and *memory reclamation*. A memory management system with good memory utilization will waste as little space as possible in the full amount of memory it must manage. If the current memory has lots of small chunks of unused memory, finding a chunk to accommodate an allocation request is costly, so the allocation is slow. On the other hand, if the memory manager does not have to search every available chunk of unused memory to find a free block to satisfy an allocation request, the allocation is quite fast. Thus, memory utilization and speed of allocation are at direct odds with each other. Regardless of which factor you decide on, it is possible that enough free memory exists to satisfy an allocation request, but the problem is that the required amount is not contiguous. The full memory block is said to be *externally fragmented*. You have a few options in responding to the allocation request. The easiest, but most undesirable, is to simply fail and return a null pointer. Another possibility is to borrow memory from the global heap, but this defeats the requirement of having a memory budget. Yet another possibility, one that is desirable but comes with some cost, is to reorganize the free memory and package it into a contiguous block, after which the allocation request is granted. The reorganization is referred to as *memory compaction*, or *garbage collection*.

19.3.1 ALLOCATION USING SEQUENTIAL-FIT METHODS

The full memory block is a contiguous collection of bytes. Sub-blocks of contiguous bytes are either *free* for allocation or *used* because they were already allocated and are currently in use by the application. The free blocks are maintained as a doubly linked list. Hypothetical block structures are listed next. The status flag, Used, is listed as a

Boolean in order to keep the pseudocode simple. In practice you can make this a 1-bit quantity, using the high-order bit of the size variable.

```
class HeaderBlock
{
public:
    bool Used;
    unsigned int Size;
    Block* Prev;
    Block* Next;
};

class FooterBlock
{
public:
    bool Used;
    unsigned int Size;
};
```

It is possible to represent a doubly linked list with only a single pointer member per object. The trick is to store the exclusive-or of the addresses of the previous and next blocks. If P is the address of the previous block and N is the address of the next block, the current block stores $P \otimes N$, where $\otimes$ denotes the exclusive-or of the address treated as integer values. Two useful properties for exclusive-or are $P = (P \otimes N) \otimes N$ and $N = (P \otimes N) \otimes P$. Let L be a known address for a node in the doubly linked list, typically the starting node for a list traversal. You must also have an address S to “start” a traversal. Let this be the address of the previous node for L , say, $S = P$. To obtain the address of the next node, use $L \otimes S = (P \otimes N) \otimes P = N$.

The HeaderBlock and FooterBlock classes are intended for use by the MemoryManager, which allocates the full memory block using malloc and typecasts it to be of type Block.

```
m_memoryBudget = <integer specified by client system>;
m_fullMemory = (char*)malloc(memoryBudget);

// for convenience in allocating/deallocating
m_hsize = sizeof(HeaderBlock);
m_fsize = sizeof(FooterBlock);
m_hfsize = m_hsize + m_fsize;

HeaderBlock* header = (HeaderBlock*)m_fullMemory;
header->Used = false;
header->Size = m_memoryBudget;
```



```

FooterBlock* footer = (FooterBlock*)(m_fullMemory +
    m_memoryBudget - m_fsize);
footer->Used = false;
footer->Size = m_memoryBudget;

// The memory manager stores the free list of blocks as a
// doubly linked cyclic list.
header->Prev = header;
header->Next = header;

// the first free block to examine when allocating
m_freeBlock = (HeaderBlock*)m_fullMemory;

```

The header information is stored in the first part of the full memory block. The information is duplicated at the end of the block. As blocks are split during allocation, each sub-block has header information associated with it. This status and size information is repeated at the end of the block. These are convenient for coalescing adjacent sub-blocks during deallocation; the details are presented later in this discussion.

Memory allocation is described by the following pseudocode. The function `SearchByPolicy` iterates over the list of free blocks, starting with `m_freeBlock`, searching for a block with sufficient memory to satisfy the request. You get to specify a policy for the search, of which a few are discussed later.

```

char* MemoryManager::Allocate (unsigned int requestedSize)
{
    HeaderBlock* header = SearchByPolicy(requestedSize);
    if (header == NULL)
    {
        // What to do when the allocation request fails?
        return NULL;
    }

    unsigned int Size = header->Size;
    FooterBlock* footer = header + Size - m_fsize;
    if (Size == requestedSize + m_hfsize)
    {
        // The block is an exact fit for the requested size.

        // the allocated memory from the caller's perspective
        char* allocated = header + m_hsize;
        header->Used = true;
        footer->Used = true;
    }
}

```

```

// Detach the block from the free list.
if (header->Next == header)
{
    // This is the only block on the free list.
    m_freeBlock = 0;
}
else
{
    m_freeBlock->Prev->Next = m_freeBlock->Next;
    m_freeBlock->Next->Prev = m_freeBlock->Prev;
    m_freeBlock = m_freeBlock->Next;
}
return allocated;
}

// The free block has more storage than is needed for the
// allocation request. We need to split it into two blocks,
// but because of header/footer space requirements, the
// free block needs to be sufficiently large.
if (Size >= requestedSize + 2*m_hfsize)
{
    // the allocated memory from the caller's perspective
    char* allocated = header + m_hsize;

    // Split the block into a used block and a free block.
    HeaderBlock* usedHeader = header;
    FooterBlock* usedFooter = header + m_hsize + requestedSize;
    HeaderBlock* freeHeader = usedFooter + m_fsize;
    FooterBlock* freeFooter = footer;
    usedHeader->Used = true;
    usedFooter->Used = true;
    freeHeader->Used = false;
    freeFooter->Used = false;

    // Update the block sizes to reflect the splitting.
    unsigned int usedSize = requestedSize + m_hfsize;
    freeHeader->Size = header->Size - usedSize;
    usedHeader->Size = usedSize;

    // The used block is not part of a list structure. The
    // free block inherits the previous/next pointers from
    // the used block.

```

```

        freeHeader->Prev = header->Prev;
        freeHeader->Next = header->Next;

        m_freeBlock = freeHeader;

        return allocated;
    }

    // The free block has more storage than is needed for the
    // allocation request, but there is not enough storage to
    // split the block into two sub-blocks, each with a header
    // and a footer. Allocate as in the exact-fit case, but
    // with the understanding that a small amount of memory is
    // wasted (the application is unaware this memory exists).
    // This is called "internal fragmentation."

    // the allocated memory from the caller's perspective
    char* allocated = header + m_hsize;
    header->Used = true;
    footer->Used = true;

    // Detach the block from the free list.
    if (header->Next == header)
    {
        // This is the only block on the free list.
        m_freeBlock = 0;
    }
    else
    {
        m_freeBlock->Prev->Next = m_freeBlock->Next;
        m_freeBlock->Next->Prev = m_freeBlock->Prev;
        m_freeBlock = m_freeBlock->Next;
    }
    return allocated;
}

```

Figure 19.1 shows the free list before and after splitting of the current block.

Memory deallocation is described by the following pseudocode. Attempts are made to coalesce the deallocated block with memory-adjacent free blocks.

```

void MemoryManager::Deallocate (char* toDeallocate)
{
    // Assert: toDeallocate is not NULL.
    HeaderBlock* header = toDeallocate - m_hsize;

```

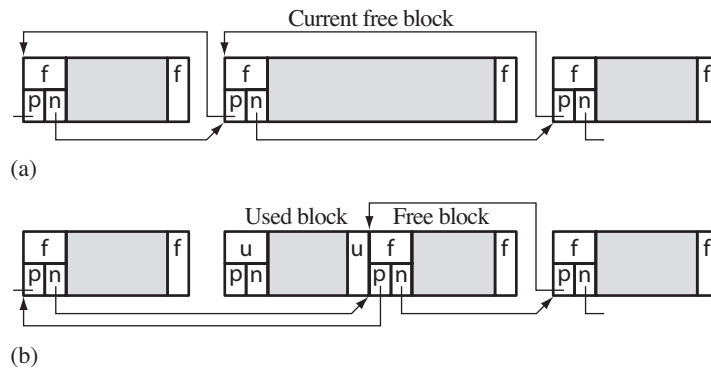


Figure 19.1 The free list of memory blocks. (a) The free block before splitting. (b) The used block and free block after splitting.

```
FooterBlock* footer = header + header->Size - m_fsize;
header->Used = false;
footer->Used = false;

if (m_freeBlock == NULL)
{
    // The free list is empty; just add the block to it.
    // The header and footer sizes are already correct.
    header->Prev = header;
    header->Next = header;
    m_freeBlock = header;
    return;
}

// Check if the left memory-adjacent neighbor is free.
// If so, coalesce the two blocks.
FooterBlock* leftFooter = header - m_fsize;
if (leftFooter->Used == false)
{
    HeaderBlock* leftHeader = leftFooter -
        leftFooter->Size - m_hsize;
    leftHeader->Size += header->Size;
    leftFooter->Size = leftHeader->Size;
```

```

    if (leftHeader->Next == leftHeader)
    {
        // This block is the only free one remaining on the free list.
        return;
    }

    // Remove the coalesced block from the free list. This
    // makes the current block appear to be used, which sets
    // up for coalescing with the right block and/or for
    // adding the coalesced block back to the free list.
    header = leftHeader;
    header->Next->Prev = header->Prev;
    header->Prev->Next = header->Next;
}

// Check if the right memory-adjacent neighbor is free.
// If so, coalesce the two blocks.
HeaderBlock* rightHeader = header + header->Size;
if (rightHeader->Used == false)
{
    FooterBlock* rightFooter = rightHeader +
        rightHeader->Size - m_fsize;
    header->Size += rightHeader->Size;
    rightFooter->Size = header->Size;

    header->Prev = rightHeader->Prev;
    header->Next = rightHeader->Next;

    if (header->Prev == header)
    {
        // This block is the only free one remaining on the free list.
        m_freeBlock = header;
        return;
    }

    // Remove the coalesced block from the free list. This
    // makes the current block appear to be used, which sets
    // up for adding the coalesced block back to the free list.
    header->Next->Prev = header->Prev;
    header->Prev->Next = header->Next;
}

// Add the block to the free list.
m_freeBlock->Prev->Next = header;

```

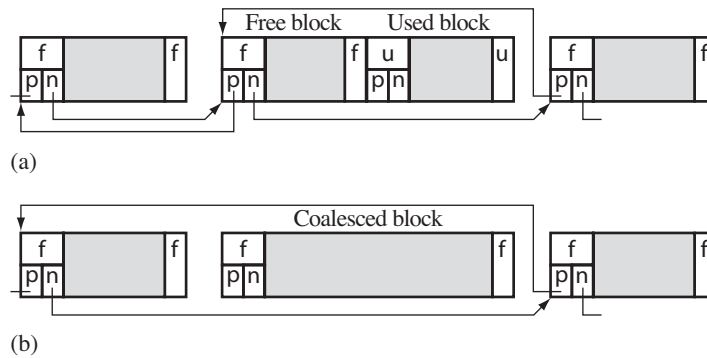


Figure 19.2 Coalescing the used block with the left free block. (a) Before coalescing. (b) After coalescing.

```

header->Prev = m_freeBlock->Prev;
header->Next = m_freeBlock;
m_freeBlock->Prev = header;
}

```

The Boolean Used fields in the header and footer are set to false, so all processing in this function is with blocks marked as “free.”

Figure 19.2 shows the situation before and after coalescing the used block with the left free block. The coalesced block is removed from the free list. This allows the attempt to coalesce with the right free block, making it appear as if the current block is used. If the block to be deallocated does not have a left free neighbor, then the block is also ready to attempt to coalesce with the right free block.

Figure 19.3 shows the situation before and after coalescing the used block with the right free block. The coalesced block is removed from the free list. This supports adding the block to the free list. If the block to be deallocated had neither a left free nor a right free neighbor, it is ready to be added to the free list.

The mechanics for updating the free list are straightforward. The heart of the matter, though, is the policy for searching that is encapsulated by the function `SearchByPolicy` in `MemoryManager::Allocate`. A few standard policies are as follows:

- *First fit.* The free list of blocks is traversed until a block is found that is large enough to satisfy the allocation request. Of all the policies, this one tends to have the minimum amount of search time, but can choose quite large blocks when in fact a better-fitting one exists.

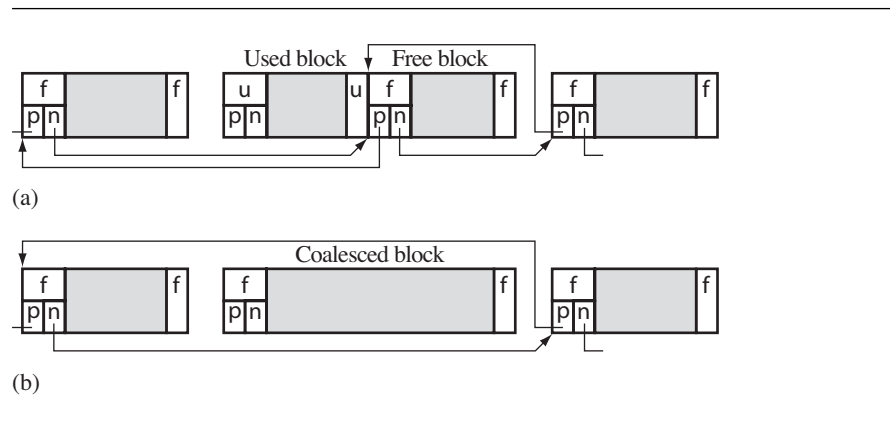


Figure 19.3 Coalescing the used block with the right free block. (a) Before coalescing. (b) After coalescing.

- *Best fit.* The free list of blocks is traversed until either an exact-fitting block is found or until the entire list has been traversed. In the latter case, the smallest block able to satisfy the request is chosen. This can be quite expensive because of having to traverse the entire list, but in exchange you get good memory utilization.
- *Optimal fit.* This is an approach that is designed to be a compromise between the first-fit and best-fit policies. The free list of blocks is traversed. A sample of n free blocks able to satisfy the allocation request is created, and the size of the best-fitting block in the sample is tracked. After the sample is created, the free list is traversed further until a block is found that is a better fit for the allocation request than the best-fitting block in the sample. This “optimal” block is used to satisfy the request. The concept has quite a bit of statistics to back it up. The result is that a certain percent of the time (dependent on n) you will actually choose the block produced by the best-fit policy. If the optimal-fit policy fails to find a suitable block after the sample is built, the method falls back to the best-fit policy. An amortized analysis shows that the optimal-fit policy generally performs better than the best-fit policy (on average, of course).
- *Worst fit.* The largest block on the free list is chosen. The idea is that over time, the number of small blocks on the free list will be lower than what other policies produce.

Regardless of the policy selected, the general rule of thumb is that the sequential-fit methods have good memory utilization but slow memory allocation because of the free-list traversals.

- EXERCISE** Implement a memory manager that uses sequential-fit methods. Implement the four policies discussed here and devise some experiments to compare their performances.
- 19.1

19.3.2 ALLOCATION USING BUDDY-SYSTEM METHODS

A different approach to memory allocation is the *buddy system*. The goal is to improve on sequential-fit methods by having faster allocation, hopefully not leading to a significant reduction in memory utilization.

The ideas are illustrated for *binary buddy systems*. The full memory pool has 2^m storage locations for some user-specified m . The memory blocks used to satisfy allocation requests are required to have sizes that are powers of 2. The maximum block size is, of course, 2^m . The minimum block size must be large enough to store any header and footer information as well as what the user has requested for the allocation. The memory manager maintains a set of doubly linked lists for tracking free blocks, one for each power-of-2 block size. That is, the doubly linked list `FreeList[p]` contains free blocks that are all of size 2^p .

When a block of size n is requested for allocation (n includes the size of storage for headers and footers), the smallest p is computed for which $n \leq 2^p$. If `FreeList[p]` is not empty, a block on this list is used to satisfy the allocation request. If $n < 2^p$, there will be some unused memory in the block—exactly $2^p - n$ storage locations. Thus, some internal fragmentation can occur in the system.

If `FreeList[p]` is empty but `FreeList[p+1]` is not, a free block in the latter list is used to satisfy the allocation request. This block has 2^{p+1} storage locations, but we already know that $n \leq 2^p$, so there is an excess of wasted storage, at least 2^p storage locations. To avoid the waste, the block is split into two sub-blocks, each with 2^p storage locations. The block of size 2^{p+1} is removed from `FreeList[p+1]`, the two sub-blocks are added to `FreeList[p]`, and the first of these sub-blocks is used to satisfy the allocation request.

Of course it is also possible that `FreeList[p+1]` is empty. Generally, the first nonempty free list is located, say, `FreeList[p+i]`, for some $i > 0$. A block in this list has size 2^{p+i} and is removed. The block is split into two sub-blocks, one of them placed on the list `FreeList[p+i-1]`. The other sub-block is further subdivided and the process repeated until you finally have added one block to each of the lists `FreeList[p]` through `FreeList[p+i-1]` and the remaining block is of size 2^p and is used to satisfy the allocation request.

When a block of size 2^{k+1} is subdivided into two sub-blocks, each of size 2^k , the sub-blocks are said to be *buddies*. The memory deallocation algorithm releases a used block of size 2^k by adding it to `FreeList[k]`. If that block's buddy is also on the free list, the two are removed from `FreeList[k]` and coalesced into a block of size 2^{k+1} , and the coalesced block is added to `FreeList[k+1]`. It is possible yet again that the buddy of this block is also on that free list. These are coalesced and the process repeated until a block reinserted into a free list does not have its buddy there.

To coalesce two blocks, it is not sufficient that they be adjacent in memory. Consider a block of 4 bytes. The splitting of the block produces two sub-blocks, each having 2 bytes. These sub-blocks are further subdivided into 1-byte units. If the bytes are numbered 0, 1, 2, and 3, the original block has bytes {0, 1, 2, 3}. The first split produces blocks {0, 1} and {1, 2}. The second split produces {0}, {1}, {2}, and {3}. The bytes 0 and 1 are buddies and the bytes 2 and 3 are buddies. However, 1 and 2 are not buddies, even though they are adjacent in memory.

Because of the binary subdivision, it is easy to identify buddies based on the memory addresses of their first bytes relative to the beginning of the full memory block. That is, think of the full memory block as having address zero. Buddies of size 2^k will have relative addresses where the first buddy (in memory-address order) has 0-valued bits in locations 0 through k , where 0 is the location of the least significant bit. The second buddy has 0-valued bits in locations 0 through $k - 1$, but a 1-valued bit in location k .

Memory allocation is described by the following pseudocode. It assumes that the `MemoryManager` class has defined a set of free lists for the various block sizes. The data member of `MemoryManager` that stores the power m of 2^m , the size of the full memory, is `m_maxPower`. The `HeaderBlock` class is used here with one minor modification: the `Size` field stores the logarithm (base 2) of the block size. That is, if the block stores 2^k units, then the `Size` field stores k and not 2^k . There is no need for the `FooterBlock` class.

```
char* MemoryManager::Allocate (unsigned int requestedSize)
{
    // Compute the smallest integer k for which the requested
    // size n satisfies n <= pow(2,k) and there is a free list
    // with blocks of size pow(2,k).
    unsigned int kmin = ceiling(Log2(requestedSize));
    unsigned int k = kmin;
    List freeList = empty_list;
    for (/**/; k <= m_maxPower; k++)
    {
        freeList = m_freeList[k];
        if (freeList is not empty)
        {
            break;
        }
    }
    if (freeList is empty or k > m_maxPower)
    {
        // The request cannot be satisfied. What to do?
        return NULL;
    }
}
```

```

// Remove a block from the free list.
HeaderBlock* header = freeList.RemoveFront();
header->Used = true;

// Subdivide the blocks until you find one of the
// minimal size to satisfy the request.
while (k > kmin)
{
    // The block needs to be split.
    k = k - 1;
    HeaderBlock* buddy = header + pow(2,k);
    buddy->Used = false;
    buddy->Size = k;

    // The list m_freeList[k] is empty to have gotten
    // here. Add buddy to this list (easy).
    m_freeList[k] = buddy;
    buddy->Prev = buddy;
    buddy->Next = buddy;
}

// At this point we know k == kmin.
char* allocated = header + m_hsize;
return allocated;
}

```

Memory deallocation is described by the following pseudocode. Attempts are made to coalesce buddies.

```

void MemoryManager::Deallocate (char* toDeallocate)
{
    // Assert: toDeallocate is not NULL.
    HeaderBlock* header = toDeallocate - m_hsize;

    // Get the buddy of this block.
    unsigned int k = header->Size; // Block size is really 2^k.
    unsigned int bitKmask = pow(2,k);
    HeaderBlock* buddy = header XOR bitKmask; // complement bit k

    while (true)
    {
        if ((buddy->Used == true) // buddy in use
            || (k == m_maxPower) // header is full memory
            || (buddy->Size != k)) // buddy free, size mismatch
        {

```

```

    {
        // The buddy is not available for coalescing.
        header->Used = false;
        m_freeList[k].InsertBlock(header);
        return;
    }

    // The buddy is available for coalescing.
    m_freeList[k].RemoveBlock(buddy);
    header = minimum(header, buddy);
    header->Used = false;
    k = k + 1;
    header->Size = k;
    bitKmask = 2*bitKmask;
    buddy = header XOR bitKmask;
}
}

```

The use of power-of-2 blocks leads to poor memory utilization because of the internal fragmentation but is relatively fast for allocation. Other methods of splitting lead to less internal fragmentation. The sizes of blocks and their two binary buddies are represented by the recurrence relation

$$S_p = 2^p, \quad S_n = S_{n-1} + S_{n-1}, \quad n > p$$

with S_n the size of the block and S_{n-1} the size of the sub-blocks. The smallest block size is S_p . Alternative buddy systems use different recurrence relations. The Fibonacci buddy system uses

$$S_n = S_{n-1} + S_{n-2}, \quad n \geq 2$$

for suitably chosen initial sizes S_0 and S_1 . A generalized Fibonacci buddy system uses

$$S_n = S_{n-1} + S_{n-k}, \quad n \geq k$$

and requires you to specify $k > 2$ and initial sizes S_0 through S_{k-1} . Yet another alternative is to support sizes 2^k and $3 \cdot 2^k$ in hopes of obtaining less internal fragmentation.

EXERCISE 19.2

Implement a memory manager that uses the binary buddy method. **Note:** The pseudocode for allocation and deallocation is based on the full memory block having starting address zero; that is, the pseudocode uses relative addressing. Your actual implementation needs to take this into account when computing addresses. ■

19.3.3 ALLOCATION USING SEGREGATED-STORAGE METHODS

The sequential-fit methods and the buddy methods both use a contiguous block of memory to manage. The blocks can be of arbitrary (and nonuniform) sizes. The mixing of block sizes is problematic in affecting the performance of the memory manager, but mixed block sizes are essential in a general-purpose system. *Segregated-storage methods* are a compromise in hopes of obtaining better performance.

A contiguous block of memory is used by the memory manager, but this block is partitioned into smaller chunks. Each chunk has its own mechanism for storage, allocation, and deallocation. In a sense, I have already suggested this for game console applications—each engine gets its own chunk of memory and each engine is responsible for managing that memory. In a general-purpose system, each chunk might represent a collection of uniform-sized blocks, which are easier to manage regarding allocation and deallocation (much like the class `new` operator does).

I will not go into the details of such methods here. Systems like these are used in memory paging systems for standard operating systems. They have also been studied extensively in the field of database management. The systems have partitioned storage, but then require some type of indexed structure on top of the partitions to allow fast access to them. The material in the preceding sections on memory management is intended to get you to think about memory budgets and the fact that building your own memory management system for a game component (graphics engine, physics engine, etc.) is a reasonable thing to do when you need really tight control over the resources on a game console.

19.3.4 MEMORY COMPACTION

Although I am certain your own applications never run out of memory (famous last words), a memory management system must deal with the improbable, but possible, failure to grant an allocation request. If you really are out of all possible memory, consider yourself short on luck. However, it is more likely that memory is available, but the free blocks are not large enough to satisfy the allocation request.

In a language such as Java, the memory management system automatically handles this problem. Moreover, you the programmer do not have to explicitly allocate or deallocate memory. The objects in Java are reference counted. Whenever you are finished referencing an object and either explicitly or implicitly release your hold on that object, if the object's reference count becomes zero, Java will make the associated memory available for later use. This might not happen the moment you release your hold on the object, but the language does allow you to tell the system to do garbage collection.

In a language such as C++, you call `new` and `delete` as needed. If you forget to delete memory at the time you no longer need it, no one will remind you of this. You are responsible for requesting allocations and deallocations. Use of the Standard Template Library (STL) might help you to some degree because the containers tend

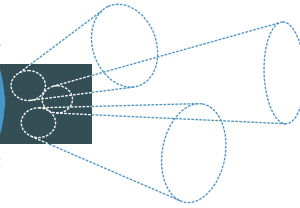
to isolate you from when memory is allocated or deallocated. However, your applications might very well be dynamically allocated STL containers, in which case you are back to managing allocation and deallocation requests. In Wild Magic, much of the responsibility for this is handled by the smart pointer and reference counting system. The `Object` class has support for this, and the initialization and termination system has components that attempt to detect *object leaks*. The `Memory` class provides support for detecting *memory leaks*. The two together give you some automatic support for memory management at a high level.

That said, the inevitability of a failed allocation request must be dealt with. This is especially true if you write your own memory manager as described previously. Reclaiming small memory blocks in order to satisfy an allocation request is called *memory compaction*.

The topics of memory compaction and garbage collection are quite large. I will not discuss the details here. You may find a good presentation of this in many data structure books and in Volume 1 of Knuth's famous series [Knu73]. The key algorithms to look into are *mark-and-sweep* methods for automatic reclamation, *incremental garbage collection* in order to amortize the cost of reclamation rather than forcing the application to wait for a long period of time when reclamation is initiated, and *copying and moving of cyclic, doubly linked lists in bounded memory*. The last topic is important when your memory usage is nearly at capacity, so you have very little free space to store temporary blocks as you move other chunks of memory around.

What you will find out about memory compaction, though, is interesting. The issue comes down to working with multilinked structures. In particular, the types of operations you work with are *traversing*, *compacting*, *copying*, *moving*, and *equality testing*. The scene graph data structures I have described in this book and have implemented in Wild Magic are in fact multilinked structures. The operations I have mentioned here all apply to scene graphs. Traversing is the most common operation you have seen. Copying and moving come into play during streaming operations, whether to memory or to disk. Streaming to disk can also be viewed as a form of compacting, but so can various scene graph compilers. Although I do not have a subsystem in the scene graph management system for equality testing, we put such a system into NetImmerse/Gamebryo. The conclusion? Scene graph management is just a particular case of memory management. If you want more details on scene graph management, read as much as you can about multilinked data structures and memory management.

In the context of a game application on a console with limited memory, your goal is to avoid having a subsystem run out of the memory budgeted to it. My preference is to trap the failure to satisfy an allocation request, and then determine if the problem is that the budget was really not large enough or if the problem really is memory fragmentation. The latter case is not what you want in the game application, so it is better to spend your time on rethinking the budget rather than on implementing a sophisticated memory compaction and garbage collection scheme.



SPECIAL EFFECTS USING SHADERS

Various shader programs are described in this chapter. Their level of complexity varies from very simple (vertex colors, single textures, lights, and materials) to very complicated (planar shadows, planar reflections, water effects). The shader programs built into the engine are found in the folder

GeometricTools/WildMagic4/Shader Programs/Cg

Some applications have their own specially constructed shader programs. The shader files are located in the application folders.

20.1 VERTEX COLORS

Applying vertex colors to a geometric object is a simple task. The Vertex.cg file contains two vertex programs:

```
void v_VertexColor3
(
    in float4      kModelPosition : POSITION,
    in float3      kModelColor : COLOR,
    out float4      kClipPosition : POSITION,
    out float3      kVertexColor : COLOR,
    uniform float4x4 WVPMatrix)
```

```

{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    // Pass through the vertex color.
    kVertexColor = kModelColor;
}

void v_VertexColor4
(
    in float4      kModelPosition : POSITION,
    in float4      kModelColor : COLOR,
    out float4     kClipPosition : POSITION,
    out float4     kVertexColor : COLOR,
    uniform float4x4 WVPMatrix)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    // Pass through the vertex color.
    kVertexColor = kModelColor;
}

```

The first program is for RGB vertex colors; the second program is for RGBA vertex colors. If you choose to use the RGBA version, you must enable alpha blending when drawing an object to which the effect is attached. You must choose the source and destination blending modes according to your needs.

As with any shader program, the input model-space positions must be transformed to clip-space (projection) coordinates using the composition of the model-to-world (W) matrix, the world-to-view (V) matrix, and the view-to-projection (P) matrix. The composition occurs in the *Renderer* code and the matrix is assigned to the shader constant registers.

In both vertex programs, the vertex colors are passed through. The rasterizer interpolates these to generate per-pixel colors. The pixel programs that are paired with the vertex programs are in the file *PassThrough.cg*:

```

void p_PassThrough3
(
    in float3 kInPixelColor : COLOR,
    out float4 kPixelColor : COLOR)
{
    kPixelColor.rgb = kInPixelColor;
    kPixelColor.a = 1.0f;
}

```

```

void p_PassThrough4
(
    in float4 kInPixelColor : COLOR,
    out float4 kPixelColor : COLOR)
{
    kPixelColor = kInPixelColor;
}

```

The program `v_VertexColor3` is paired with `p_PassThrough3`. The alpha value is not part of the inputs but must be set for the output. In this case, it is assigned a value of 1 so that the pixel color is opaque.

The program `v_VertexColor4` is paired with `p_PassThrough4`. As described in Section 3.1.10, if alpha blending is enabled, the pixel color output from the pixel program will be blended accordingly with the contents of the frame buffer (or render target that is an offscreen buffer).

The classes `VertexColor3Effect` and `VertexColor4Effect` are the wrappers for shader effects using simple vertex coloring.

20.2 LIGHTING AND MATERIALS

Lighting and materials were discussed in Section 2.6.2. Certain mathematical formulas were mentioned for a simple lighting model. Some shader programs that implement these models are found in the file `Lighting.cg`. I will describe these briefly in a moment.

For objects whose vertices all have the same color, vertex colors are overkill because of their memory usage. Instead you may use a material-only (no lighting) vertex program. My implementation is in the file `Material.cg`:

```

void v_Material
(
    in float4      kModelPosition : POSITION,
    out float4      kClipPosition : POSITION,
    out float4      kDiffuseColor : COLOR,
    uniform float4x4 WVPMatrix,
    uniform float4  MaterialDiffuse)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    // Use the material diffuse color as the vertex color.
    kDiffuseColor = MaterialDiffuse;
}

```


The program requires that you have a `Material` global state object attached to the geometric object. If you do not provide one, a default `Material` object is used. All that you need to set in the `Material` object are the diffuse (RGB) color and the alpha (A) value. The combined RGBA value is passed as the shader constant `MaterialDiffuse`. If the alpha value is smaller than 1, you must have alpha blending enabled in order to obtain semitransparent drawing.

The diffuse-plus-alpha value is passed through. The rasterizer will interpolate this and send it to each call of the pixel program. Since the diffuse-plus-alpha values are the same at every pixel, you could instead use the default vertex program, in file `Default.cg`:

```
void v_Default
(
    in float4 kModelPosition : POSITION,
    out float4 kClipPosition : POSITION,
    uniform float4x4 WVPMatrix)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);
}
```

You could then create a pixel program with uniform inputs consisting of the diffuse-plus-alpha value. This saves cycles on the GPU because only the clip-space position must be interpolated during rasterization.

In the material vertex program listed here, the shader constant `MaterialDiffuse` is one of the automatic values sent by the renderer to the graphics API shader system. You do not have to manage this value as a user-defined constant.

The class `MaterialEffect` is the wrapper for a shader effect using only material colors. Generally, material colors alone are visually unappealing. When you use materials in conjunction with lighting, you can obtain better results. The file `Lighting.cg` has four vertex programs: one for an ambient light, one for a directional light, one for a point light, and one for a spotlight. The file contains some helper functions that implement the mathematical models described in Section 2.6.2. These are

- `GetDirectionalLightFactors`. Computes the diffuse and specular coefficients for the directional light model.
- `GetPointLightFactors`. Computes the diffuse and specular coefficients for the point-light model. The inputs are different from those for the directional light factors, so a separate function is needed to compute the point-light factors.
- `GetSpotLightFactors`. Computes the diffuse, specular, and spot coefficients for the spotlight model.
- `GetAttenuation`. Computes the attenuation coefficient for point lights and spotlights. This coefficient is not relevant for ambient or directional lights.

The file has two additional vertex programs, one for two ambient lights and one for an ambient and a diffuse light. These illustrate how to obtain multilight vertex programs, making calls to the helper functions that compute the light factors. Support for multiple lights is the sole reason for factoring out the helper functions as I did.

For simple lighting with no textures, the pixel program to pair with the lighting vertex programs is `p_PassThrough4`, which is found in the file `PassThrough.cg`. The class `LightingEffect` is the wrapper for a shader effect using simple lighting.

20.2.1 AMBIENT LIGHTS

The vertex program for a single ambient light is

```
void v_L1a
(
    in float4      kModelPosition : POSITION,
    out float4      kClipPosition : POSITION,
    out float4      kVertexColor : COLOR,
    uniform float4x4 WVPMatrix,
    uniform float3   MaterialEmissive,
    uniform float3   MaterialAmbient,
    uniform float3   Light0Ambient,
    uniform float4   Light0Attenuation)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    float3 kLamb = Light0Attenuation.w*Light0Ambient;
    kVertexColor.rgb = MaterialEmissive + MaterialAmbient*kLamb;
    kVertexColor.a = 1.0f;
}
```

The shader constants are all set up automatically by the renderer. It uses the active `Material` object, presumably one attached to the geometric primitive, to look up the emissive and ambient material colors. It also uses the first light stored in index 0 of its internal array of active lights. The ambient color of the light is looked up and sent to the shader system of the graphics API. Although ambient lighting does not support attenuation, the `Light` class in *Wild Magic* allows you to adjust the intensity of the light. This value is stored as one of the attenuation constants; it is passed via the *w*-component of the 4-tuple `Light0Attenuation`.

The RGB values of the output vertex color have an ambient component, which was defined in Equation (2.77). The material emissive color is added to the result. The material's alpha channel is not processed here, so the vertex colors are opaque (an alpha of 1).

20.2.2 DIRECTIONAL LIGHTS

The vertex program for a single directional light is

```
void v_L1d
(
    in float4      kModelPosition : POSITION,
    in float3      kModelNormal : NORMAL,
    out float4     kClipPosition : POSITION,
    out float4     kVertexColor : COLOR,
    uniform float4x4 WVPMatrix,
    uniform float3  CameraModelPosition,
    uniform float3  MaterialEmissive,
    uniform float3  MaterialAmbient,
    uniform float4  MaterialDiffuse,
    uniform float4  MaterialSpecular,
    uniform float3  Light0ModelDirection,
    uniform float3  Light0Ambient,
    uniform float3  Light0Diffuse,
    uniform float3  Light0Specular,
    uniform float4  Light0Attenuation)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    float fDiff, fSpec;
    GetDirectionalLightFactors((float3)kModelPosition,kModelNormal,
        CameraModelPosition,Light0ModelDirection,MaterialSpecular.a,
        fDiff,fSpec);

    float3 kColor = MaterialAmbient*Light0Ambient;
    if (fDiff > 0.0f)
    {
        kColor += fDiff*MaterialDiffuse.rgb*Light0Diffuse;
        if (fSpec > 0.0f)
        {
            kColor += fSpec*MaterialSpecular.rgb*Light0Specular;
        }
    }

    kVertexColor.rgb = MaterialEmissive + Light0Attenuation.w*kColor;
    kVertexColor.a = MaterialDiffuse.a;
}
```

The shader constants are all set up automatically by the renderer. It uses the active Material object, the first light in the array of active lights, and the camera's model-space position. All lighting is performed in model space for efficiency, avoiding applying transformations to inputs to convert them to other coordinate systems.

The diffuse coefficient μ from Equation (2.79) and the specular coefficient γ from Equation (2.82) are computed by `GetDirectionalLightFactors`, returning the results in `fDiff` and `fSpec`, respectively. The diffuse contribution is computed according to Equation (2.78) and the specular contribution is computed according to Equation (2.81).

The vertex color is assigned the alpha value of the material. If the material alpha is smaller than 1, alpha blending must be enabled to obtain the semitransparent effect.

20.2.3 POINT LIGHTS

The vertex program for a single point light is

```
void v_Llp
(
    in float4      kModelPosition : POSITION,
    in float3      kModelNormal : NORMAL,
    out float4     kClipPosition : POSITION,
    out float4     kVertexColor : COLOR,
    uniform float4x4 WVPMatrix,
    uniform float4x4 WMatrix,
    uniform float3  CameraModelPosition,
    uniform float3  MaterialEmissive,
    uniform float3  MaterialAmbient,
    uniform float4  MaterialDiffuse,
    uniform float4  MaterialSpecular,
    uniform float3  Light0ModelPosition,
    uniform float3  Light0Ambient,
    uniform float3  Light0Diffuse,
    uniform float3  Light0Specular,
    uniform float4  Light0Attenuation)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    float fDiff, fSpec;
    GetPointLightFactors(kModelPosition.xyz,kModelNormal,
        CameraModelPosition,Light0ModelPosition,MaterialSpecular.a,
        fDiff,fSpec);
```

```

float fAttn = GetAttenuation((float3x3)WMatrix,kModelPosition.xyz,
    Light0ModelPosition,Light0Attenuation);

float3 kColor = MaterialAmbient*Light0Ambient;
if (fDiff > 0.0f)
{
    kColor += fDiff*MaterialDiffuse.xyz*Light0Diffuse;
    if (fSpec > 0.0f)
    {
        kColor += fSpec*MaterialSpecular.xyz*Light0Specular;
    }
}

kVertexColor.rgb = MaterialEmissive + fAttn*kColor;
kVertexColor.a = MaterialDiffuse.a;
}

```

Once again, the shader constants are all set up automatically by the renderer and all lighting is performed in model space for efficiency. The diffuse and specular coefficients are computed by `GetPointLightFactors` and the diffuse and specular contributions are computed, just as for directional lights. One difference, though, is the attenuation computation. Attenuation depends on the distance from the light source to the vertex position. This computation must be done in world space in the event that the model-to-world matrix has nonunit scaling. For example, if the model-to-world matrix has uniform scaling of 2, the model-space distance between the light source and the vertex position is half that of the world-space distance. Nonuniform scaling makes this even more complicated. Therefore, the model-to-world matrix must be accessible to the vertex program; it is passed as the shader constant `WMatrix`. The `GetAttenuation` function computes the vector from the model-space light source to the model-space vertex position, and then transforms it to world-space coordinates. The length of the vector is calculated and used in the calculation of the attenuation factor.

20.2.4 SPOTLIGHTS

The vertex program for a single spotlight is

```

void v_L1s
(
    in float4 kModelPosition : POSITION,
    in float3 kModelNormal : NORMAL,
    out float4 kClipPosition : POSITION,
    out float4 kVertexColor : COLOR,

```

```

uniform float4x4 WVPMatrix,
uniform float4x4 WMatrix,
uniform float3 CameraModelPosition,
uniform float3 MaterialEmissive,
uniform float3 MaterialAmbient,
uniform float4 MaterialDiffuse,
uniform float4 MaterialSpecular,
uniform float3 Light0ModelPosition,
uniform float3 Light0ModelDirection,
uniform float3 Light0Ambient,
uniform float3 Light0Diffuse,
uniform float3 Light0Specular,
uniform float4 Light0SpotCutoff,
uniform float4 Light0Attenuation)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    float fDiff, fSpec, fSpot;
    GetSpotLightFactors(kModelPosition.xyz,kModelNormal,
        CameraModelPosition,Light0ModelPosition,MaterialSpecular.a,
        Light0ModelDirection,Light0SpotCutoff.y,Light0SpotCutoff.w,fDiff,
        fSpec,fSpot);

    float fAttn = GetAttenuation((float3x3)WMatrix,kModelPosition.xyz,
        Light0ModelPosition,Light0Attenuation);

    float3 kColor = MaterialAmbient*Light0Ambient;
    if (fSpot > 0.0f)
    {
        if (fDiff > 0.0f)
        {
            kColor += (fSpot*fDiff)*MaterialDiffuse.rgb*Light0Diffuse;
            if (fSpec > 0.0f)
            {
                kColor += (fSpot*fSpec)*MaterialSpecular.rgb*Light0Specular;
            }
        }
    }

    kVertexColor.rgb = MaterialEmissive + fAttn*kColor;
    kVertexColor.a = MaterialDiffuse.a;
}

```

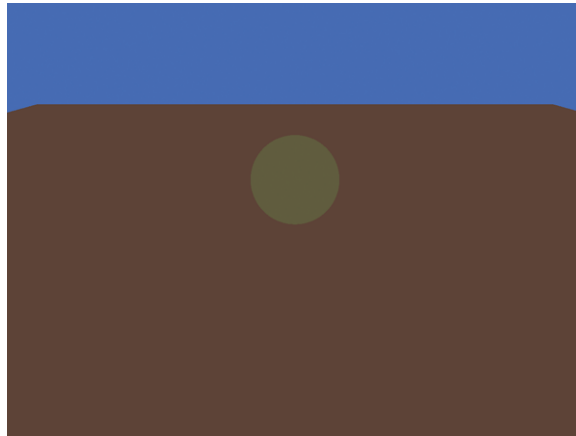


Figure 20.1 A plane and sphere illuminated by a single ambient light.

The comments made about point lights apply as well to spotlights. The spotlight vertex program does additional work, computing the spot coefficient of Equation (2.84).

The sample application `Lighting` contains a plane and a sphere. The initial configuration has the default effect attached to the objects, so they show up as magenta. You can attach lights to the scene by pressing the lowercase keys `a` (ambient light), `d` (directional light), `p` (point light), or `s` (spotlight). Lights are detached by pressing the uppercase equivalents. Up to eight lights are supported. Wireframe is toggled using the `w` key. You may select the active light by pressing the keys `0` through `7`. Once a light is active, you can decrease its intensity by pressing the lowercase `i` key or you can increase its intensity by pressing the uppercase `I` key. If the active light is a spotlight, you can adjust its cone angle by pressing `c` (decrease angle) or `C` (increase angle). You can also adjust its spot exponent by pressing `e` (decrease the exponent by one-half its value) or `E` (increase the exponent by doubling its value).

By pressing the `a` key, a single ambient light is attached to the scene. Figure 20.1 shows the result. The lighting is somewhat dark and the image appears flat in color.

Now press the `d` key so that an ambient light and a directional light are attached to the scene. Figure 20.2 shows the result. The lighting uses multipass rendering. If you want single-pass lighting, you can write a shader to do so. The `Lighting.cg` file already has a shader in it for single-pass lighting for an ambient and a directional light. You need to compile this vertex program to `wmsp` files.

Remove the directional light by pressing the `D` key and add a point light by pressing `p`. Figure 20.3 shows the result. Remove the point light by pressing the `P` key and add a spotlight by pressing `s`. Figure 20.4 shows the results. The spot angle is

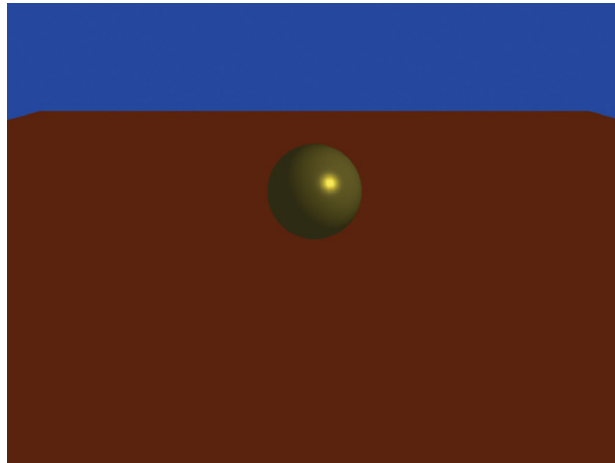


Figure 20.2 A plane and sphere illuminated by an ambient light and a directional light.

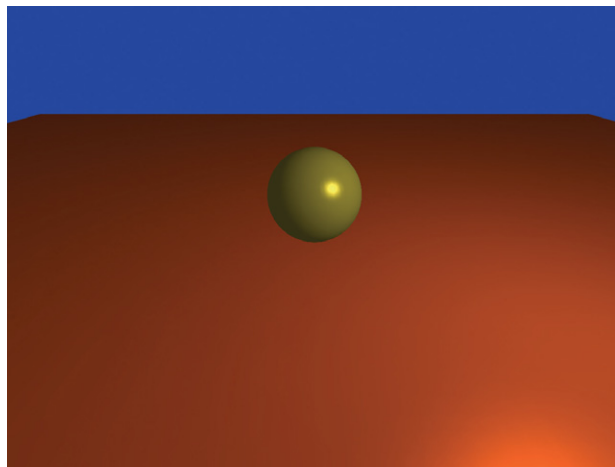


Figure 20.3 A plane and sphere illuminated by an ambient light and a point light.

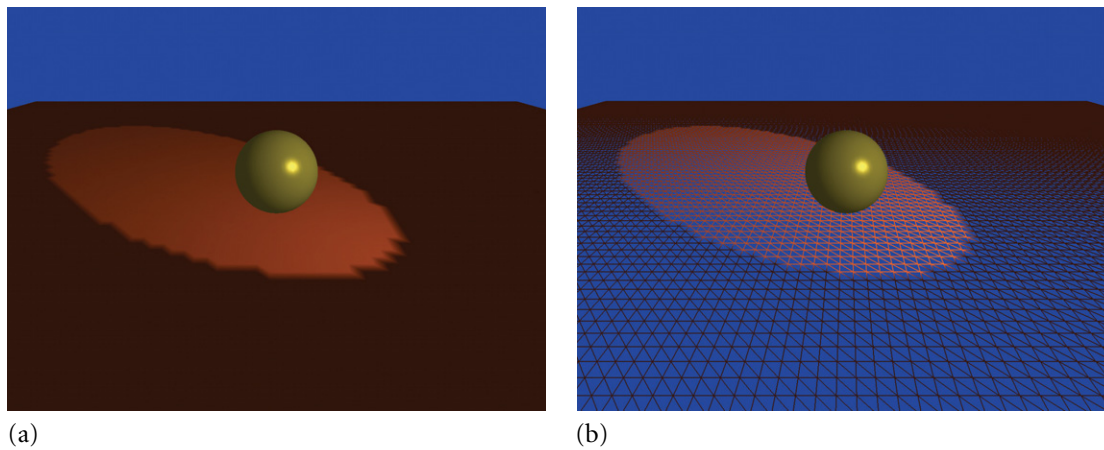


Figure 20.4 (a) A plane and sphere illuminated by an ambient light and a spotlight with spot exponent 1. (b) A wireframe view of the ground.

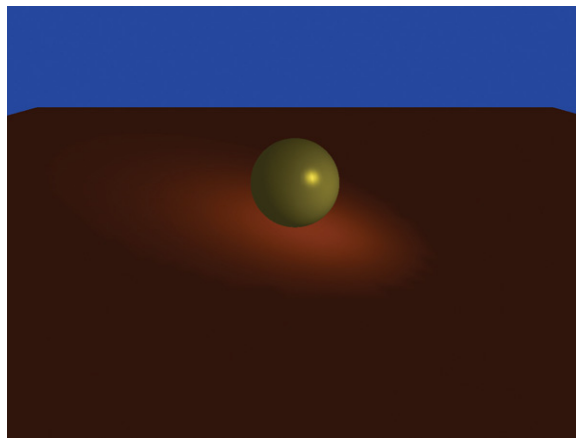


Figure 20.5 A plane and sphere illuminated by an ambient light and a spotlight with spot exponent 32.

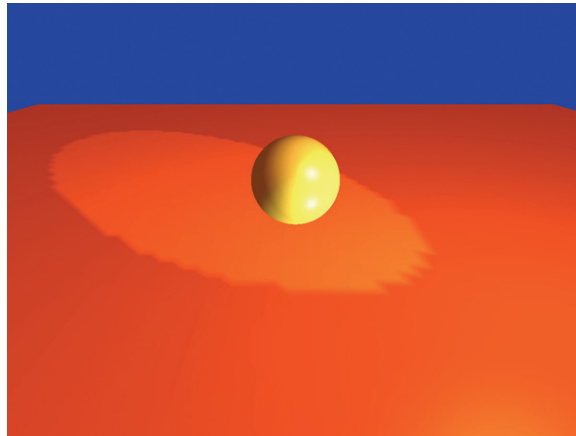


Figure 20.6 A plane and sphere illuminated by two ambient lights, two directional lights, two point lights, and two spotlights.

$\pi/8$ radians and the spot exponent is 1. Notice the jaggedness around the brightly lit region. The tessellation of the plane is coarse enough to cause the jaggedness. You can eliminate some of this effect by choosing a larger spot exponent. Figure 20.5 shows the spotlighting with an exponent of 32.

Finally, Figure 20.6 shows the scene lit by eight lights: two ambient lights, two directional lights, two point lights, and two spotlights. The display is washed out due to the accumulation of so many lights. A small amount of dynamic lighting is reasonable to use, but it is probably better to rely on texturing effects and full-screen image-space effects to obtain better realism. More sophisticated lighting models are also better to use than the standard models, but they come at the cost of more GPU cycles.

20.3 TEXTURES

Applying a single texture to a geometric object is a simple task. The `Texture.cg` file contains two programs that are used to apply a 2D texture to a geometric primitive:

```
void v_Texture
(
    in float4      kModelPosition : POSITION,
    in float2      kInBaseTCoord : TEXCOORD,
    out float4     kClipPosition : POSITION,
```

```

    out float2      kOutBaseTCoord : TEXCOORD0,
    uniform float4x4 WVPMatrix)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    // Pass through the texture coordinate.
    kOutBaseTCoord = kInBaseTCoord;
}

void p_Texture
(
    in float2      kBaseTCoord : TEXCOORD0,
    out float4      kPixelColor : COLOR,
    uniform sampler2D BaseSampler)
{
    // Sample the texture image.
    kPixelColor = tex2D(BaseSampler,kBaseTCoord);
}

```

The vertex program assumes that 2D texture coordinates have been associated with the vertices of the geometric primitive. The coordinates are passed through by the program. The rasterizer interpolates these, producing per-pixel texture coordinates and passing them to the pixel program. The pixel's texture coordinates are passed to the sampler associated with the texture, and the sampled color is returned as the pixel color.

The Cg compiler associates texture units with the samplers. The Program parser reads this information and creates Wild Magic data structures containing this information, as well as other texture unit information. The `PixelShader` object stores Texture objects that contain information for setting up the texture unit, including filter type and texture coordinate wrap types. The setup occurs automatically for you, but you are responsible for associating images with the texture and for choosing the filter and wrap types. The Cg files with extension `.fx` are designed for you to place this information in a file, load it, and use the Cg Runtime to handle the setup. Wild Magic has its own system, just for illustration, but does not yet have an equivalent to an `.fx` file.

The class `TextureEffect` is the wrapper for a shader effect using a single 2D texture. If you want basic vertex and pixel programs for a 3D texture, you need to choose more suggestive names for the programs. The naming conventions become important also when you have multiple textures and multiple blending modes for combining them, the topic of the next section.

20.4 MULTITEXTURES

Most good-looking special effects use multiple textures on a single object. For example, you might want a brick texture for a wall, but then blend into it a light map to give the wall the appearance of being lit without actually resorting to the more expensive dynamic lighting. Another example is to have a window with a base texture to give it a glassy tint (perhaps a bluish tint), a decal texture to make it appear as if it has a bullet hole with fracture lines around the hole, and an environment map to make it appear as if the window is reflecting what is behind the observer.

You can easily achieve these effects with shader programs that handle multiple textures in a single pass. To write the classes corresponding to all the possibilities of blending two through n textures, each blend operation chosen from a multitude of source and destination blending modes, would be a time-consuming adventure. This is exactly the situation where you need to think about *shader stitching*, creating new shaders as combinations of old ones and compiling them on the fly. Wild Magic does not yet have a shader stitching system. For now, the class `MultitextureEffect` supports multitexturing, but only a few shader programs are provided to go with it. You can write additional ones as needed, using the naming convention supported by the class. This was described in Section 4.6.1.

The sample applications

```
GeometricTools/WildMagic4/SampleGraphics/Multitextures
GeometricTools/WildMagic4/SampleGraphics/Multieffects
```

illustrate drawing a geometric primitive with two textures, supporting three different blending operations. The `Multitextures` sample uses single-pass multitexturing using an object of class `MultitextureEffect`. The `Multieffects` sample shows the same geometric primitive and textures, but the result is obtained by attaching two `TextureEffect` objects to the primitive and using a multipass drawing operation. Visually, you get the same results, but the frame rates are different. On my test machines, the single pass drawing is about 20 to 25 percent faster.

Figure 20.7 shows the two texture images that are applied to a square. The default blending used in the application is multiplicative mode; the final texture is the product of the two input textures. Figure 20.8 shows the result. If you press the `n` key, the blending switches to hard additive mode; the final texture is the sum of the two input textures. Figure 20.9 shows the result. If you press the `n` key again, the blending switches to soft additive mode; the final texture is the sum of the two input textures. If C_0 and C_1 are the two texture colors at a single pixel, the soft addition is $(1 - C_1) \circ C_0 + C_1$, where the products and sums are componentwise. Figure 20.10 shows the result.

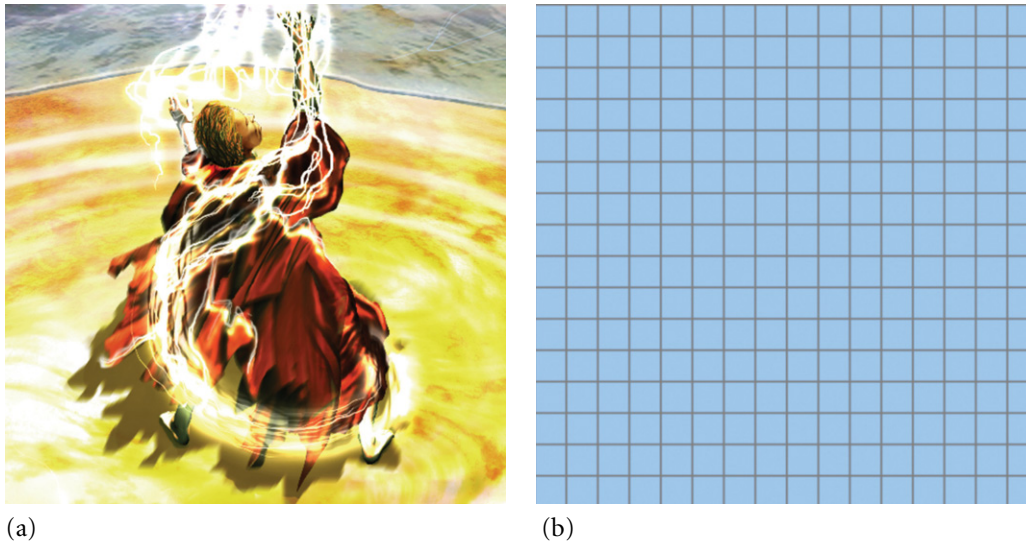


Figure 20.7 The two texture images used in the Multitextures and Multieffects sample applications. (a) The primary texture image. (b) The secondary texture image.

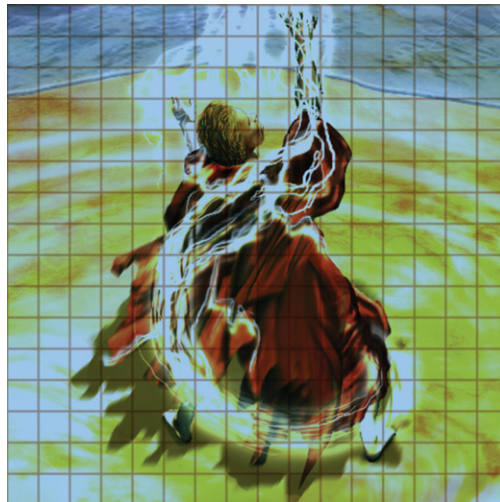


Figure 20.8 The two textures are multiplied during rendering.



Figure 20.9 The two textures are added during rendering. Notice how washed out the result is. This happens because the sum of colors at many pixels exceeds the upper bound on the color channels, so they are clamped to white.

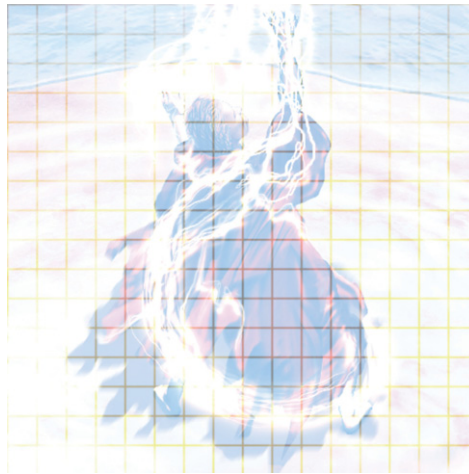


Figure 20.10 The two textures are blended according to $(1 - C_1) \circ C_0 + C_1$. The result is not washed out as it was for hard additive mode. No clamping is necessary for soft additive mode.

20.5 BUMP MAPS

Bump-mapping is the generic term for a texturing mechanism that allows you to provide normal vectors on a per-pixel basis and is used to control per-pixel lighting. The texture that stores the normal vector information is called a *normal map*. The red, green, and blue channels of the texture colors are encodings of unit-length normal vectors $\mathbf{N} = (n_0, n_1, n_2)$. The associated color is $\mathbf{C} = (r, g, b) = (1 + n_0, 1 + n_1, 1 + n_2)/2$ with components in $[0, 1]$. The pixel program maps such colors to normal vectors to be used in the per-pixel lighting.

20.5.1 GENERATING NORMAL MAPS

The original texture image is converted to a gray-scale image. The intensities are treated as heights. The normal vectors to this surface are estimated by using finite difference approximations. In the sample application, the original texture image shows a collection of bricks. Figure 20.11 shows the texture image and a gray-scale mapping of it.

Figure 20.12 shows a rendering of the height field generated by the gray-scale image. The camera has been placed close to the height field so you can see the height variation. The normal vectors are the gradients of the height field, which happen to be normal to the surface.

The idea is that in regions of approximately constant intensity, the object to be textured is flat. The height field normals point directly upward from the surface. The bricks themselves have this property. In regions with significant changes in intensity, the height field normals point up to a 90-degree angle from the up direction (they point “off to the sides”). The portions of the mortar immediately adjacent to the bricks have this property. The texture is modulated by the dot product between the light direction and the normal direction at a point on the surface.

The normal map is generated for the gray-scale image according to the following. Think of the gray-scale image as $H(x, y)$, where (x, y) are the continuous-valued texture coordinates. The texture images are defined for left-handed (x, y) , but we want to process the height field in right-handed (x, y, z) . To preserve the order of x and y , the right-handed system is obtained by negating z , which flips the direction of one of the basis vectors for the coordinate system. Thus, we look at the graph of the function $z = -H(x, y)$. Define $F(x, y, z) = H(x, y) + z$; the height field is defined implicitly by $F(x, y, z) = 0$. From calculus, normal vectors are the gradient of F :

$$\nabla F = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right) = \left(\frac{\partial H}{\partial x}, \frac{\partial H}{\partial y}, 1 \right)$$

The normal vectors are chosen to be $\mathbf{N} = \nabla F / |\nabla F|$.

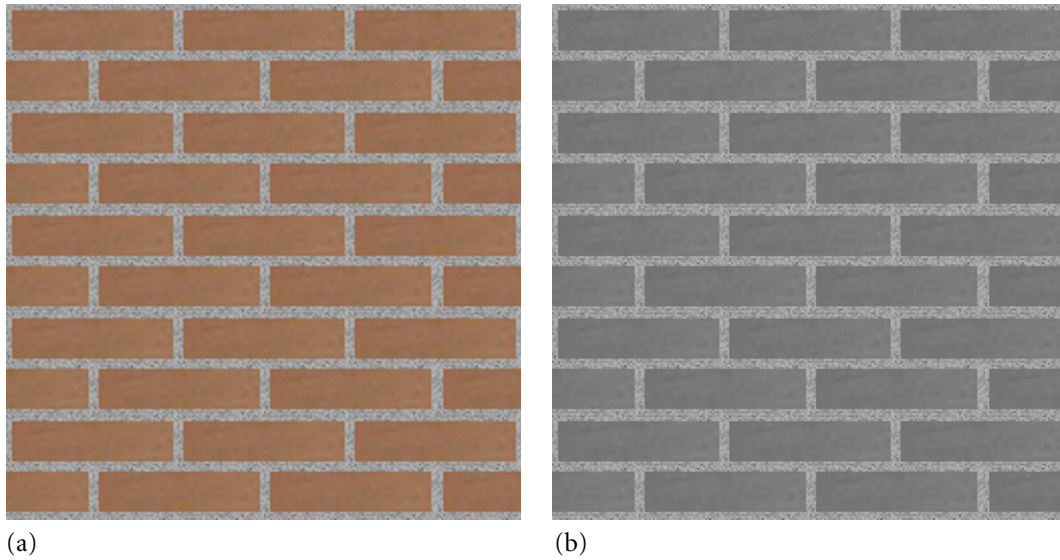


Figure 20.11 (a) The original texture image shows a collection of bricks. (b) A gray-scale mapping of the image. The intensities are computed by $I = 0.2125R + 0.7154G + 0.0721B$, where R is the red channel, G is the green channel, and B is the blue channel.

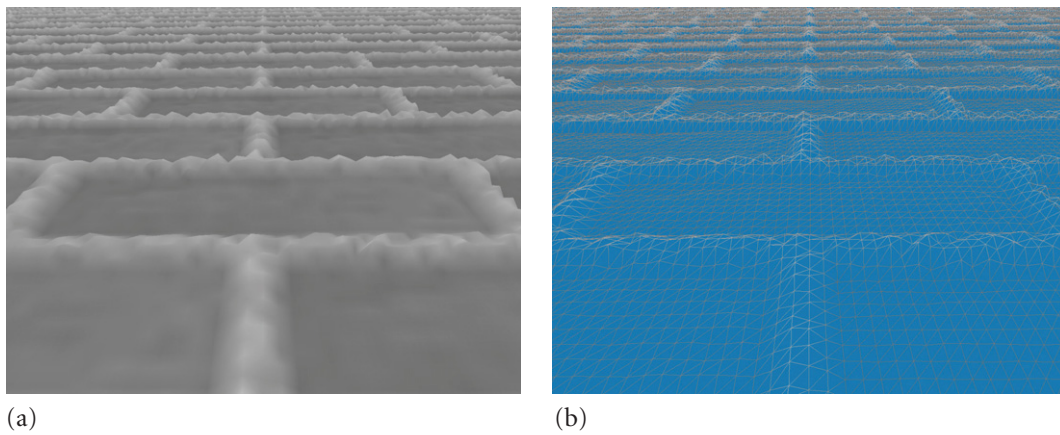


Figure 20.12 (a) A close-up rendering of the height field generated by the gray-scale image of Figure 20.11 (b). (b) The same rendering but in wireframe mode.

Finite difference approximations are used to estimate the derivatives:

$$\frac{\partial H}{\partial x} \doteq \frac{H(x + \Delta x, y) - H(x - \Delta x, y)}{2\Delta x}$$

and

$$\frac{\partial H}{\partial y} \doteq \frac{H(x, y + \Delta y) - H(x, y - \Delta y)}{2\Delta y}$$

In pixel units, we may as well choose $\Delta x = 1$ and $\Delta y = 1$. Special handling must occur at the image boundaries $x = 0$, $y = 0$, $x = w - 1$, and $y = h - 1$, where the image has width w and height h . One method is to use wraparound, treating (w, y) as $(0, y)$ and (x, h) as $(x, 0)$. This is reasonable when the texture coordinates are set to repeat, but if you use texture coordinates that are strictly inside the domain of the image, the wraparound is fine to use. You can also use one-sided estimates at the boundary; for example, the one-sided x -derivative estimate at $x = 0$ is

$$\frac{\partial H(0, y)}{\partial x} \doteq \frac{H(\Delta x, y) - H(0, y)}{\Delta x}$$

Similar formulas exist for the three other boundary derivative estimates.

The `Tools` folder has a project, `CreateNormalMap`, which implements the normal map construction from a color bitmap. The program allows you to scale the height field to give you more control over the normals. The default is to scale the gray-scale values to be in $[-1, 1]$. You can select a multiplier σ to change this to $[-\sigma, \sigma]$. As you decrease the value of σ to be smaller than 1, the visual effect is to reduce the variation in the normal vectors. If you have a sharp edge, say, at the pixels just between brick and mortar, a large variation in normals produces a very dark or very bright transition. With smaller values of σ , the transition is less dark or less bright.

The normal map computed for the bricks texture of Figure 20.11 is shown in Figure 20.13.

EXERCISE 20.1

Assuming a normal map was constructed using finite difference approximations as mentioned here, reconstruct the height field from it. The answer will depend on whether the approximations used wraparound or one-sided estimates at the boundaries. You will have to assume that you know the height for one pixel, so you may as well choose the height at pixel $(0, 0)$ to be zero. ■

20.5.2 GENERATING TANGENT-SPACE INFORMATION

The variation I have implemented in the `BumpMaps` sample application uses *tangent-space lighting*. A coordinate system is defined at each point $\mathbf{P}$ on a surface. If $\mathbf{N}$ is the unit-length normal vector obtained from the gradient of the height field, then it is possible to compute two unit-length tangent vectors $\mathbf{T}$ and $\mathbf{B}$. The first vector

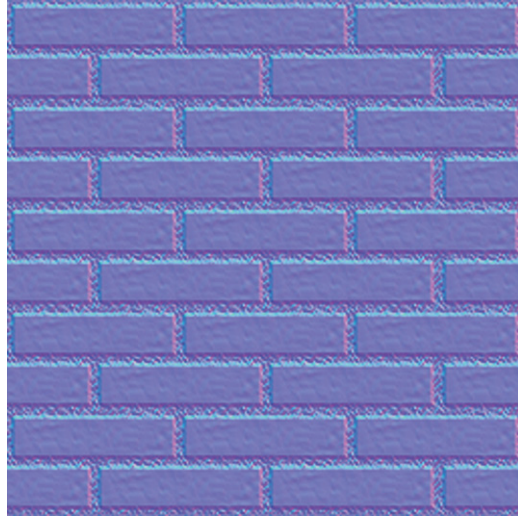


Figure 20.13 The normal map for the bricks texture. This map was generated by the CreateNormalMap tool.

is usually just called the *tangent* and the second vector is called the *bitangent*.¹ It is important that the tangent and bitangent vary as smoothly as possible. For a surface with the topology of a square, this is always possible. For a surface with the topology of a sphere, it is impossible to guarantee global continuity.² If we can compute a tangent $\mathbf{T}$, the bitangent is computed using a cross product, $\mathbf{B} = \mathbf{N} \cdot \mathbf{T}$. The ordered set $\{\mathbf{T}, \mathbf{B}, \mathbf{N}\}$ is a right-handed orthonormal set.

The negative of the light direction vector at the surface position $\mathbf{P}$ is $-\mathbf{D}$ and is in world coordinates. It is transformed to the model-space coordinates of the geometric primitive to be drawn; call this vector $\mathbf{L}$. This vector is represented in the coordinate system at $\mathbf{P}$:

$$\mathbf{L} = (\mathbf{L} \cdot \mathbf{T})\mathbf{T} + (\mathbf{L} \cdot \mathbf{B})\mathbf{B} + (\mathbf{L} \cdot \mathbf{N})\mathbf{N}$$

1. Some people call $\mathbf{B}$ the *binormal*. This is incorrect terminology as far as differential geometers are concerned. A binormal is defined for curves in space; in particular, it is part of the *Frenet frame* for a curve. A bitangent is defined for surfaces; the coordinate system is called the *Darboux frame*.
2. The lack of global continuity of a unit-length tangent vector field is a well-known problem in topology. If you have a continuous tangent vector field defined on a sphere, there must be at least one point on the sphere at which the tangent is the zero vector. Sometimes an analogy is made to combing a “hairy billiard ball.” If you have a ball with hair sticking straight out of it everywhere, and if you attempt to comb the hair to be flat (tangent to the ball), your intuition should tell you that at some point you have to part the hair away from it in all directions.

Although all three dot products can be used in advanced shader effects, in the simple bump-mapping sample application, I only use the component $\mathbf{L} \cdot \mathbf{N}$ to modulate the color at $\mathbf{P}$.

To generate a light vector at each point on the surface, we will generate a light vector at each triangle mesh vertex, map it to an RGB value and store it in a vertex color array, and let the rasterizer interpolate the vertex colors in the usual manner. To have the light vector vary smoothly from vertex to vertex, we need to have a parameterization of the surface and transform the light vector into the coordinates relative to the parameterization. However, the triangle mesh was most certainly generated without such a coordinate system in mind. The texture coordinates themselves may be thought of as inducing a parameterization of the surface. Each point (x, y, z) has a texture coordinate (u, v) , so you may think of the surface parametrically as $(x(u, v), y(u, v), z(u, v))$. Now we do not actually know the functions for the components. All we know are sample values at the vertices. We can obtain a continuous representation using the following. Consider a triangle with vertices $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$ and with corresponding texture coordinates (u_0, v_0) , (u_1, v_1) , and (u_2, v_2) . Any point on the triangle may be represented as

$$\mathbf{P}(s, t) = \mathbf{P}_0 + s(\mathbf{P}_1 - \mathbf{P}_0) + t(\mathbf{P}_2 - \mathbf{P}_0)$$

where $s \geq 0$, $t \geq 0$, and $s + t \leq 1$. The texture coordinate corresponding to this point is similarly represented as

$$\begin{aligned} (u, v) &= (u_0, v_0) + s((u_1, v_1) - (u_0, v_0)) + t((u_2, v_2) - (u_0, v_0)) \\ &= (u_0, v_0) + s(u_1 - u_0, v_1 - v_0) + t(u_2 - u_0, v_2 - v_0) \end{aligned}$$

Abstractly, we have a surface defined by $\mathbf{P}(s, t)$, where s and t depend implicitly on two other parameters u and v . The problem is to estimate a tangent vector relative to u or v . We will estimate with respect to u , a process that involves computing the rate of change of $\mathbf{P}$ as u varies, namely, the partial derivative $\partial \mathbf{P} / \partial u$. Using the chain rule from calculus:

$$\frac{\partial \mathbf{P}}{\partial u} = \frac{\partial \mathbf{P}}{\partial s} \frac{\partial s}{\partial u} + \frac{\partial \mathbf{P}}{\partial t} \frac{\partial t}{\partial u} = (\mathbf{P}_1 - \mathbf{P}_0) \frac{\partial s}{\partial u} + (\mathbf{P}_2 - \mathbf{P}_0) \frac{\partial t}{\partial u}$$

Now we need to compute the partial derivatives of s and t with respect to u . The equation that relates s and t to u and v is written as a system of two linear equations in two unknowns:

$$\begin{bmatrix} u_1 - u_0 & u_2 - u_0 \\ v_1 - v_0 & v_2 - v_0 \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} u - u_0 \\ v - v_0 \end{bmatrix}$$

Inverting this leads to

$$\begin{bmatrix} s \\ t \end{bmatrix} = \frac{1}{(u_1 - u_0)(v_2 - v_0) - (u_2 - u_0)(v_1 - v_0)} \begin{bmatrix} v_2 - v_0 & -(u_2 - u_0) \\ -(v_1 - v_0) & u_1 - u_0 \end{bmatrix} \begin{bmatrix} u - u_0 \\ v - v_0 \end{bmatrix}$$

Computing the partial derivative with respect to u produces

$$\begin{aligned} \begin{bmatrix} \partial s / \partial u \\ \partial t / \partial u \end{bmatrix} &= \frac{1}{(u_1 - u_0)(v_2 - v_0) - (u_2 - u_0)(v_1 - v_0)} \begin{bmatrix} v_2 - v_0 & -(u_2 - u_0) \\ -(v_1 - v_0) & u_1 - u_0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ &= \frac{1}{(u_1 - u_0)(v_2 - v_0) - (u_2 - u_0)(v_1 - v_0)} \begin{bmatrix} v_2 - v_0 \\ -(v_1 - v_0) \end{bmatrix} \end{aligned}$$

Combining this into the partial derivative for $\mathbf{P}$, we have

$$\begin{aligned} \frac{\partial \mathbf{P}}{\partial u} &= \frac{(v_2 - v_0)(\mathbf{P}_1 - \mathbf{P}_0) - (v_1 - v_0)(\mathbf{P}_2 - \mathbf{P}_0)}{(u_1 - u_0)(v_2 - v_0) - (u_2 - u_0)(v_1 - v_0)} \\ &= \frac{(v_1 - v_0)(\mathbf{P}_2 - \mathbf{P}_0) - (v_2 - v_0)(\mathbf{P}_1 - \mathbf{P}_0)}{(v_1 - v_0)(u_2 - u_0) - (v_2 - v_0)(u_1 - u_0)} \end{aligned} \quad (20.1)$$

which is used as an estimate of the tangent vector $\mathbf{T}$.

The actual implementation for computing $\mathbf{T}$ using Equation (20.1) must handle degenerate cases. The triangle might be needlelike, in which case the denominator in Equation (20.1) might be too small, causing numerical problems. It is also possible that the u -component of the texture coordinate does not vary sufficiently, making it appear as a constant. In this case, $\partial \mathbf{P} / \partial u = \mathbf{0}$, a degeneracy. These cases are handled in the sample bump-mapping application.

20.5.3 THE SHADER PROGRAMS

The vertex and pixel programs are listed next.

```
float3 MapFromUnit (float3 kVector)
{
    // Map [0,1] to [-1,1].
    return 2.0f*kVector - 1.0f;
}
```

```

void v_SimpleBumpMap
(
    in float4      kModelPosition : POSITION,
    in float3      kInLightDir : COLOR,
    in float2      kInBaseTCoord : TEXCOORD0,
    in float2      kInNormalTCoord : TEXCOORD1,
    out float4     kClipPosition : POSITION,
    out float3     kOutLightDir : COLOR,
    out float2     kOutBaseTCoord : TEXCOORD0,
    out float2     kOutNormalTCoord : TEXCOORD1,
    uniform float4x4 WVPMatrix)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    // Pass through the parameters.
    kOutBaseTCoord = kInBaseTCoord;
    kOutNormalTCoord = kInNormalTCoord;
    kOutLightDir = kInLightDir;
}

void p_SimpleBumpMap
(
    in float3      kLightDir : COLOR,
    in float2      kBaseTCoord : TEXCOORD0,
    in float2      kNormalTCoord : TEXCOORD1,
    out float4     kPixelColor : COLOR,
    uniform sampler2D BaseSampler,
    uniform sampler2D NormalSampler)
{
    float3 kLDir = MapFromUnit(kLightDir);
    float3 kNDir = MapFromUnit(tex2D(NormalSampler,kNormalTCoord).rgb);
    float fDot = saturate(dot(kLDir,kNDir));
    float3 kBaseColor = tex2D(BaseSampler,kBaseTCoord).rgb;
    kPixelColor.rgb = fDot*kBaseColor;
    kPixelColor.a = 1.0f;
}

```

The vertex program does the usual step of computing the clip-space position. It passes through all the other parameters. The pixel program has access to the (negative of the) light direction, but in tangent-space coordinates. The normal vectors are looked up in the normal map texture. Because both vectors are stored as RGB values with components in $[0, 1]$, they must be uncompressed to vectors with components

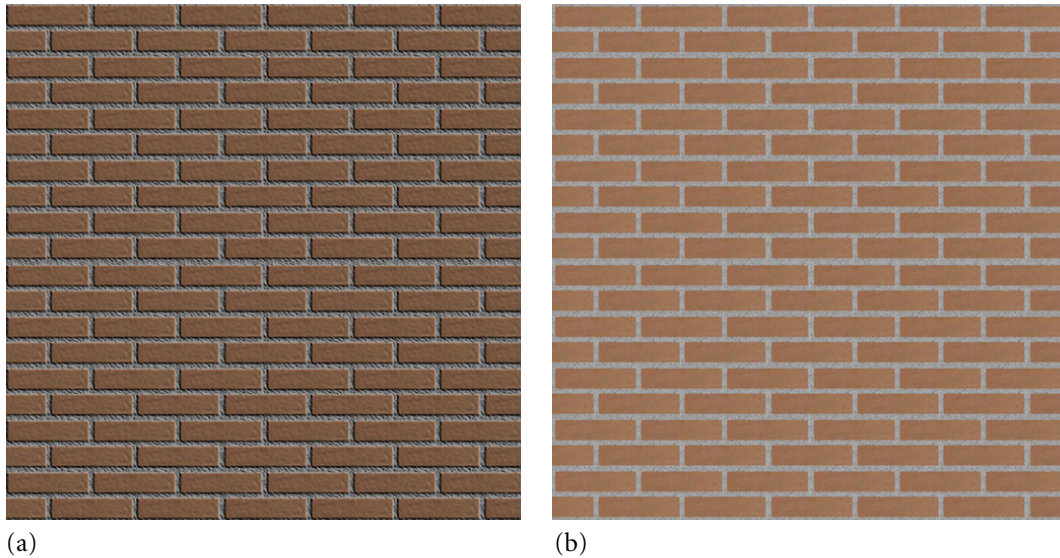


Figure 20.14 (a) A bump-mapped square. (b) The same rendering with only the base texture and no bump-mapping.

in $[-1, 1]$. The dot product of the two vectors is computed and used to modulate the base texture color.

Figure 20.14 shows two images, one with bump-mapping enabled and one with only the base texture and no bump-mapping. Notice how the bump-mapping does give the appearance that there is geometric variation in the rendering, even though the object really is planar.

To illustrate that the tangent-space coordinate system construction works correctly, the same brick texture was applied to a torus. Figure 20.15 shows the bump-mapped torus and the torus with only the base texture and no bump-mapping. There is some variation in the lighting that gives you an impression that the bump-mapped torus has some geometric variation. The impression is more pronounced when you move closer to the torus. Figure 20.16 illustrates this.

The vertex program `v_SimpleBumpMap` just passes through the (negative of the) light direction vector. Each vertex has a unit-length light vector assigned to it and compressed as RGB values. The light vectors are passed through for the rasterizer to interpolate them. The interpolated values are passed to the pixel program. The problem with this approach is that the interpolated values are not guaranteed to correspond to unit-length vectors. For example, suppose the light vector at one vertex is $(1, 0, 0)$ and at an adjacent vertex is $(0, 1, 0)$. The RGB compressed values

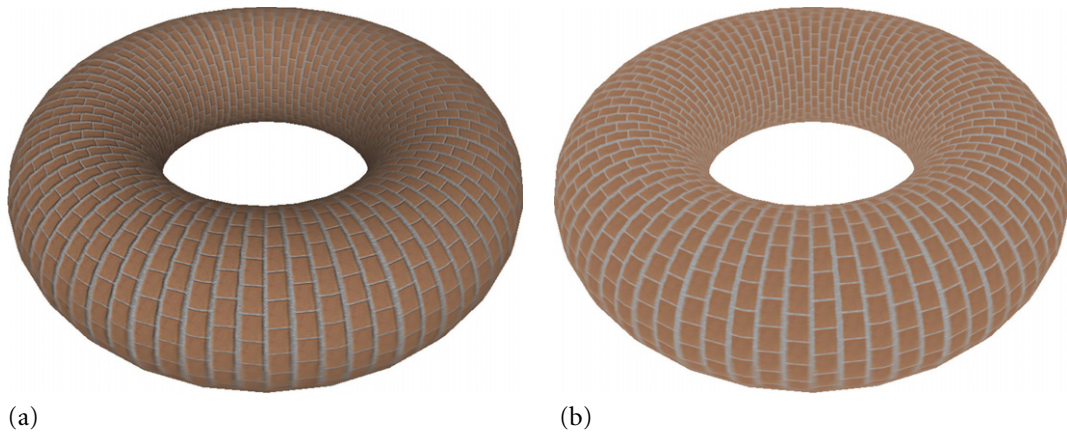


Figure 20.15 (a) A bump-mapped torus. (b) The same rendering with only the base texture and no bump-mapping.

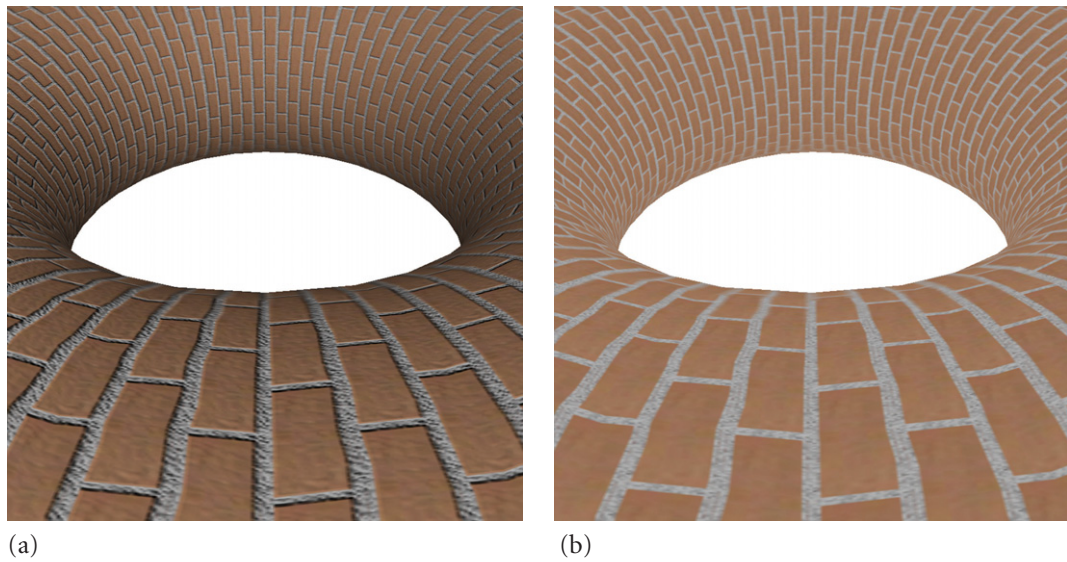


Figure 20.16 (a) A close-up view of the bump-mapped torus. (b) The same rendering with only the base texture and no bump-mapping.

are $(1, 1/2, 1/2)$ and $(1/2, 1, 1/2)$, respectively. At a pixel midway between the vertices, the interpolated RGB value is $(3/4, 3/4, 1/2)$, so the uncompressed vector is $2(3/4, 3/4, 1/2) - (1, 1, 1) = (1/2, 1/2, 0)$, which is not unit length.

A fix to this problem is to use what is called a *normalization cube map*. Cube maps are discussed in Section 20.8 and are essentially six 2D square textures that conceptually represent a texture cube. A 3D vector is used as a lookup into a cube map, a process that identifies the face intersected by the ray with the input vector direction and does a 2D texture lookup using the point that is the intersection of the ray and that face. The textures are typically a rendering of the environment surrounding the object, but in the case of normalization cube maps, the textures represent unit-length vectors. If the aforementioned ray corresponds to a non-unit-length vector $\mathbf{V}$, then the texture lookup gives you $\mathbf{V}/|\mathbf{V}|$ (in RGB compressed format). Well, almost. If you use a nearest-filter mode, you get a unit-length result. If you use bilinear filtering, four unit-length vectors are bilinearly interpolated, the result not necessarily unit length. However, what you do get should be quite close to unit length as long as your face textures are sufficiently large.

After the discussion of cube maps in Section 20.8, you will see an exercise asking you to modify the bump-mapping pixel program and the sample application to use a normalization cube map.

20.6 GLOSS MAPS

A *gloss map* is a texture that is used to modulate the specular lighting on a surface. This gives the surface a shininess in some places, as if those places reflected more specular light than other places. To achieve this effect, the shader programs must handle the specular lighting separately from the ambient and diffuse lighting.

The GlossMaps sample application is a simple illustration of gloss maps. The application uses a directional light for lighting. An RGBA texture is attached to the geometric primitive. The RGB component is modulated by the emissive-ambient-diffuse lighting. The A (alpha) component is used to modulate the specular lighting, the result added into the modulated RGB component. The vertex program is

```
void v_GlossMap
(
    in float4      kModelPosition : POSITION,
    in float3      kModelNormal : NORMAL,
    in float2      kModelTCoord : TEXCOORD0,
    out float4     kClipPosition : POSITION,
    out float3     kEADColor : COLOR,
    out float2     kTCoord : TEXCOORD0,
    out float3     kSpecularColor : TEXCOORD1,
    uniform float4x4 WVPMatrix,
    uniform float3  CameraModelPosition,
```



```

uniform float3  MaterialEmissive,
uniform float3  MaterialAmbient,
uniform float4  MaterialDiffuse,
uniform float4  MaterialSpecular,
uniform float3  Light0ModelDirection,
uniform float3  Light0Ambient,
uniform float3  Light0Diffuse,
uniform float3  Light0Specular)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    // Compute the specular color for the lit material.
    float fDiff, fSpec;
    GetDirectionalLightFactors((float3)kModelPosition,kModelNormal,
        CameraModelPosition,Light0ModelDirection,MaterialSpecular.a,
        fDiff,fSpec);

    kEADColor = MaterialEmissive + MaterialAmbient*Light0Ambient;
    kSpecularColor = 0.0f;
    if (fDiff > 0.0f)
    {
        kEADColor += fDiff*MaterialDiffuse.rgb*Light0Diffuse;
        if (fSpec > 0.0f)
        {
            kSpecularColor += fSpec*MaterialSpecular.rgb*Light0Specular;
        }
    }

    // Pass through the texture coordinate.
    kTCoord = kModelTCoord;
}

```

You should recognize the portions of the code related to lighting with a directional light; see Section 20.2. The difference is that the specular color is computed separately. The vertex program outputs the emissive-ambient-diffuse color, `kEADColor`, and the specular color, `kSpecularColor`. The texture coordinate is just passed through.

The pixel program implements the blending equation

$$(r_f, g_f, b_f) = (r_t, g_t, b_t) \circ (r_e, g_e, b_e) + a_t(r_s, g_s, b_s)$$

where (r_f, g_f, b_f) is the final color, (r_t, g_t, b_t) is the texture color, (r_e, g_e, b_e) is the emissive-ambient-diffuse color, and (r_s, g_s, b_s) is the specular color. The pixel program is

```

void p_GlossMap
(
    in float3      kEADColor : COLOR,
    in float2      kTCoord : TEXCOORD0,
    in float3      kSpecularColor : TEXCOORD1,
    out float4     kPixelColor : COLOR,
    uniform sampler2D BaseSampler)
{
    float4 kTextureColor = tex2D(BaseSampler,kTCoord);
    kPixelColor.rgb = kTextureColor.rgb*kEADColor +
        kTextureColor.a*kSpecularColor;
    kPixelColor.a = 1.0f;
}

```

The application has two squares that can be rotated simultaneously. A directional light and a material are attached to the scene, thus affecting both squares. One square has no effects attached to it and is only lit using the material colors. The other square has a gloss map attached to it. The texture image has all white RGB values, but the alpha values are 0 in the background and 1 on the pixels that lie in a text string “Magic.” As you rotate the squares, you see that the first square has a certain specular color to it. The second square has the same color but only in the region covered by the text string, giving it a glossy look. Figure 20.17 shows a couple of snapshots

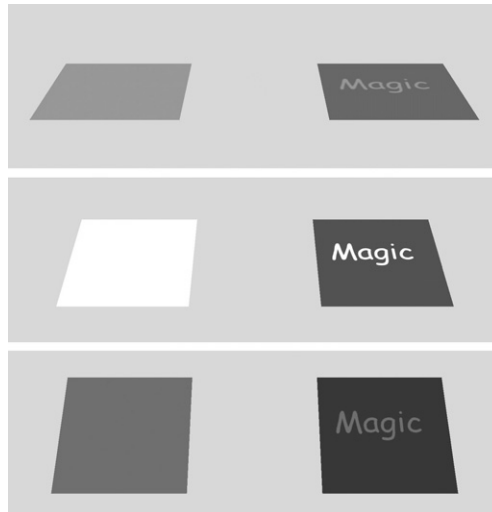


Figure 20.17 Three different rotations of the squares. The left square has lighting only via a material. The right square has a gloss map attached to it.

of the squares. The up axis has the direction $(0, 1, 0)$. The light is directional with direction $(0, -1, 0)$. When the squares are rotated to fully face the camera, both squares become completely black since the light no longer influences the visible surfaces.

20.7 SPHERE MAPS

A *sphere map* is a texture drawn on a surface that gives the appearance of the surface reflecting the environment around it. We need to assign texture coordinates to the geometric object to which the sphere map is attached. These depend on the eye point's location and the object's location and orientation. Figure 20.18 illustrates how a texture coordinate is assigned to a point on the surface.

The image to be reflected is in a coordinate space (the sphere map space) with the y -axis pointing down, the x -axis pointing to the right, and the z -axis as shown in Figure 20.18 (b). The image domain is $|x| \leq 1$ and $|y| \leq 1$. The (x, y) -coordinates are left-handed to match what we are used to for texture images. The sample application, SphereMaps, chooses this to be in view space, but with the swap from right-handed to left-handed coordinates. Figure 20.18 (a) shows the eye point E , an object with a position P and corresponding unit-length normal N , the unit-length direction of view $V = (P - E)/|P - E|$, and the reflection of V through the normal N , all in world-space coordinates. The reflection in world-space coordinates is

$$R = E - 2(N \cdot E)N$$

In view space, the normal vector N is mapped to h and the reflected vector R is mapped to r , as shown in Figure 20.18 (b).

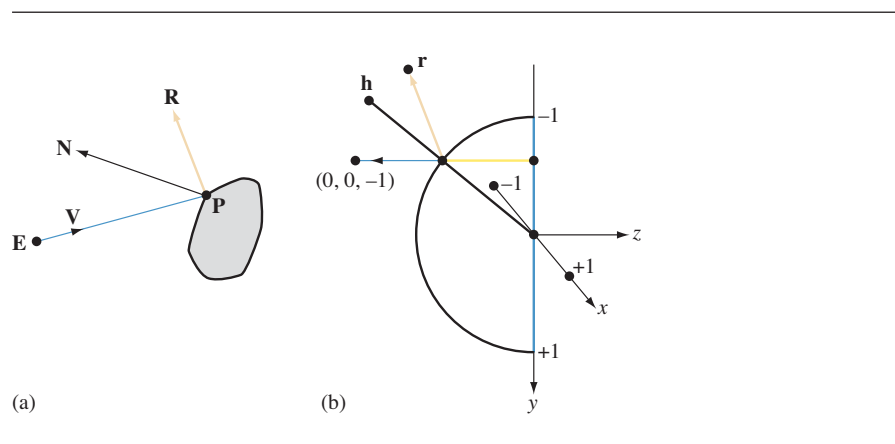


Figure 20.18 The mapping of a texture coordinate to a point on an object using sphere-mapping.

A sphere of radius 1 and centered at the view-space origin (0, 0, 0) is shown in Figure 20.18 (b). Also shown is the reflection vector $\mathbf{r}$ and the normal vector $\mathbf{h}$. The direction vector for view space is (0, 0, 1). The figure shows the negation of this positioned at the reflected vector on the sphere. The normal vector is projected onto the image plane (the vertical blue line segment); the orange line segment is the direction of projection. The black endpoint of the orange segment shown on the image plane gives you a pair (x, y), which is the texture coordinate assigned to the position $\mathbf{P}$ of the original object.

The view-space normal is the normalized average of (0, 0, -1) and $\mathbf{r} = (r_x, r_y, r_z)$:

$$\mathbf{h} = \frac{(r_x, r_y, r_z - 1)}{\sqrt{r_x^2 + r_y^2 + (r_z - 1)^2}}$$

The first two components are in $[-1, 1]$, so they must be mapped to $[0, 1]$ by adding 1 and dividing by 2. Thus, the texture coordinates are

$$x = \frac{r_x}{2\sqrt{r_x^2 + r_y^2 + (r_z - 1)^2}} + \frac{1}{2}, \quad y = \frac{r_y}{2\sqrt{r_x^2 + r_y^2 + (r_z - 1)^2}} + \frac{1}{2}$$

The shader programs for the sample application are

```
void v_SphereMap
(
    in float4      kModelPosition : POSITION,
    in float3      kModelNormal : NORMAL,
    out float4      kClipPosition : POSITION,
    out float2      kBaseTCoord : TEXCOORD0,
    uniform float4x4 WVPMatrix,
    uniform float4x4 WVMatrix)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition, WVPMatrix);

    // Transform the normal from model space to eye space.
    float3 kESNormal = normalize(mul(kModelNormal, (float3x3)WVMatrix));

    // Calculate the eye direction (in eye space).
    float3 kEyeDirection = normalize(mul(kModelPosition, WVMatrix).xyz);

    // Calculate the reflection vector.
    float3 kReflection = reflect(kEyeDirection, kESNormal);

    // Calculate the texture coordinates.
    float fOmRz = kReflection.z - 1.0f;
```

```

float fInvM = 1.0f/sqrt(kReflection.x*kReflection.x +
    kReflection.y*kReflection.y + fOmRz*fOmRz);
kBaseTCoord = 0.5f*(kReflection.xy*fInvM + 1.0f);
}

void p_SphereMap
(
    in float2      kBaseTCoord : TEXCOORD0,
    out float4      kPixelColor : COLOR,
    uniform sampler2D SphereMapSampler)
{
    kPixelColor = tex2D(SphereMapSampler,kBaseTCoord);
}

```

The vertex program maps the model-space normal to view-space coordinates. The eye location in view space is (0, 0, 0), so $\mathbf{P} - \mathbf{E}$ in world space is mapped to $\mathbf{P}'$ in view space, where $\mathbf{P}'$ is the mapping from the model-space position to view-space coordinates. This vector is normalized to obtain $\mathbf{V}$, the direction of view from the eye to the vertex. The reflection vector is computed and the texture coordinates are generated according to the previous discussion. The pixel program just uses the texture coordinates to look up the texture color in the image to be reflected. Figure 20.19 shows the image to be reflected and its rendering onto a torus.



(a)



(b)

Figure 20.19 (a) The image to be reflected by sphere-mapping. (b) The rendering of a torus to reflect the image.

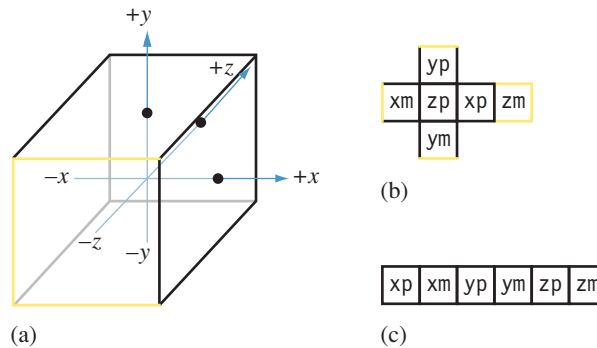


Figure 20.20 (a) The cube of textures. The images themselves are generated from a camera at the center of the cube with a 90-degree field of view. Six renderings are made, one per face. Notice that a *left-handed coordinate system* is used for (x, y, z) . (b) The cube is cut along its edges and unfolded. The label x_p indicates that face is the one in the $+x$ direction from the cube center. The other labels have similar interpretations. (c) The texture images are laid out in memory in the manner shown.

20.8 CUBE MAPS

A better form of environment mapping than using sphere maps is using *cube maps*. The environment is rendered to six faces of a cube. The resulting images are used as textures to be reflected by objects in the scene. Figure 20.20 shows the layout of the texture images and how they relate to the 3D setting.

A vector $\mathbf{V}$ is computed from the camera's world position to the world position for a vertex. This is the direction of view along which the observer sees that vertex. Just as we did for sphere maps, the direction of view is reflected. The reflection ray, when positioned at the origin of the cube, intersects the cube in some face. That point of intersection has a 2D coordinate relative to the plane of the face. This coordinate is used to look up a color in the texture image associated with that face. It is important to set up the texture images correctly. OpenGL and Direct3D have cube map samplers that do the lookups for you. The Wild Magic software renderer has its own cube map sampler. Figure 20.21 shows the left-handed coordinate systems associated with each face of the cube. The faces are numbered according to the memory layout for the texture images. Face 0 goes with $+x$; face 1 goes with $-x$; face 2 goes with $+y$; face 3 goes with $-y$; face 4 goes with $+z$; and face 5 goes with $-z$.

The cube map lookup takes the vector $\mathbf{V}$ and determines the face of the cube its ray intersects. The vector is not guaranteed to be unit length. The face it intersects is easily determined by locating the component with the largest magnitude. The logic is

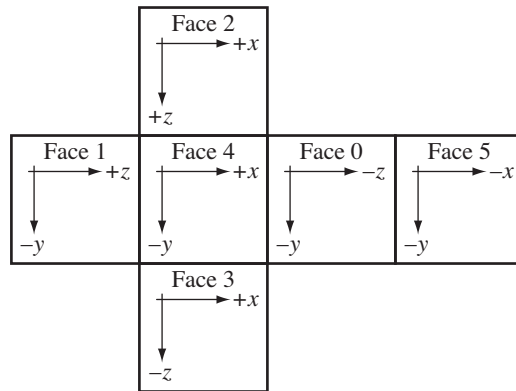


Figure 20.21 The left-handed coordinate systems for the faces of the cubes.

```

Vector3 V = (x,y,z);
float max = |x|;
int face = (x > 0 ? 0 : 1);
if (|y| > max)
{
    max = |y|;
    face = (y > 0 ? 2 : 3);
}
if (|z| > max)
{
    max = |z|;
    face = (z > 0 ? 4 : 5);
}

```

The vector components are divided by the largest magnitude, max. The component of largest magnitude becomes 1 or -1 . The other two components are in $[-1, 1]$. They are mapped to $[0, 1]$ for the actual texture coordinates:

```

float invMax = 1/max;
float u, v; // texture coordinates
switch (face)
{
case 0: // +x face, left-handed planar coordinates (-z,-y)
    (u,v) = ((1,1) + (-z,-y)/max)/2;
case 1: // -x face, left-handed planar coordinates (+z,-y)
    (u,v) = ((1,1) + (+z,-y)/max)/2;

```

```

case 2: // +y face, left-handed planar coordinates (+x,+z)
    (u,v) = ((1,1) + (+x,+z)/max)/2;
    break;
case 3: // -y face, left-handed planar coordinates (+x,-z)
    (u,v) = ((1,1) + (+x,-z)/max)/2;
case 4: // +z face, left-handed planar coordinates (+x,-y)
    (u,v) = ((1,1) + (+x,-y)/max)/2;
case 5: // -z face, left-handed planar coordinates (-x,-y)
    (u,v) = ((1,1) + (-x,-y)/max)/2;
}

```

The sample application CubeMaps is an illustration of cube maps. The environment itself is a cube whose faces have textures with the face names; for example, the $+x$ face has a texture with f_{xp} . The scene has a sphere whose vertices include positions, normals, and colors. The vertex colors are randomly generated in shades of blue and green. The sphere also has a cube map effect attached to it in order for it to reflect its environment. Figure 20.22 shows two screen captures from the sample.

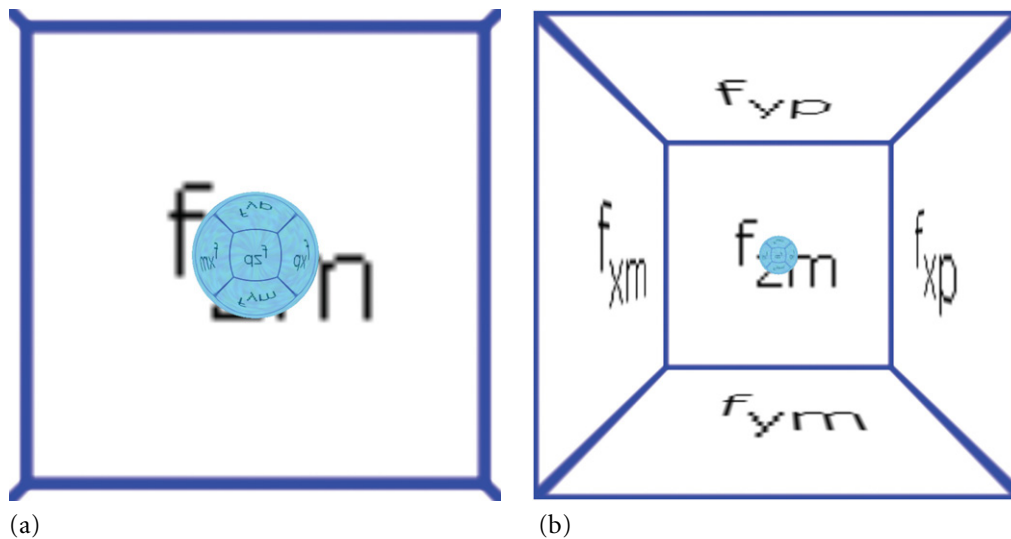


Figure 20.22 (a) The rendering when the application first runs. The camera is close to the sphere in the center of the environment. (b) The rendering when the camera is farther away from the sphere. You can see that the image on the sphere is the reflection of the walls of the cube.

The sample application also has a dynamic update system. If the environment changes, the scene is re-rendered to create the cube maps. Of course, the objects that will receive the reflection are also rendered. If one reflecting object reflects another, you will not see the environment map on the reflected object.

EXAMPLE
20.1

Create a cube map whose texture RGB values are compressed unit-length vectors. If $\mathbf{V}$ generates (s, t) on one of the faces, and if $\mathbf{U} = \mathbf{V}/|\mathbf{V}| = (u_0, u_1, u_2)$, the texture RGB color is $C(s, t) = ((1, 1, 1) + (u_0, u_1, u_2))/2$. This is called a *normalization map*. Given a vector $\mathbf{V}$, a lookup into the normalization map gives you its normalized equivalent. This is designed to avoid expensive inverse square root operations in the pixel program when the program receives non-unit-length vectors. Modify the BumpMaps pixel shader and application to use a normalization map. ■

20.9 REFRACTION

A beam of light travels through two semitransparent media that are separated by a planar surface. The speed of light through a medium depends on the density of that medium. The difference in densities at the planar surface causes the light to slightly change direction as it crosses from one medium into the other. This bending effect is called *refraction*. The planar surface typically has reflective properties. A portion of the light is reflected, the other portion refracted. Figure 20.23 depicts a beam of light that is reflected and refracted.

The incoming light has direction $\mathbf{L}$ and the unit-length outward surface normal is $\mathbf{N}$. A unit-length normal to the plane spanned by $\mathbf{L}$ and $\mathbf{N}$ is $\mathbf{L} \times \mathbf{N}/|\mathbf{L} \times \mathbf{N}|$. The

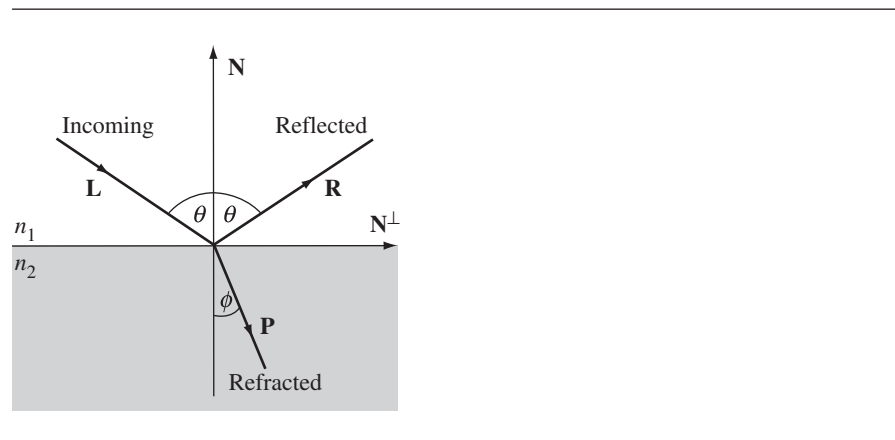


Figure 20.23 Reflection and refraction of a light beam traveling through two media.

vector $\mathbf{N}^\perp$ is also unit length and is defined by

$$\mathbf{N}^\perp = \mathbf{N} \times \frac{\mathbf{L} \times \mathbf{N}}{|\mathbf{L} \times \mathbf{N}|}$$

The angles of incidence and reflection are both θ . Notice that $\cos \theta = -\mathbf{L} \cdot \mathbf{N}$. Some basic trigonometry and linear algebra will convince you that

$$\mathbf{L} = (-\cos \theta)\mathbf{N} + (\sin \theta)\mathbf{N}^\perp$$

The reflected light has direction

$$\mathbf{R} = (\cos \theta)\mathbf{N} + (\sin \theta)\mathbf{N}^\perp = \mathbf{L} + (2 \cos \theta)\mathbf{N} = \mathbf{L} - 2(\mathbf{N} \cdot \mathbf{L})\mathbf{N} \quad (20.2)$$

The refraction angle ϕ is different than the reflection angle θ because of the difference in densities of the media. The actual amount of bending depends on the ratio of densities, or equivalently, on the ratio of speeds through the media. Physicists assign an equivalent measure called the *index of refraction*. In Figure 20.23, the index of refraction of the first medium is n_1 and the index of refraction of the second medium is n_2 . If v_1 and v_2 are the speeds of light through the media, then $n_1/n_2 = v_2/v_1$. The indices of refraction, the reflection angle, and the refraction angle are related by *Snell's law*:

$$n_1 \sin \theta = n_2 \sin \phi$$

It follows that

$$\cos \phi = \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 \sin^2 \theta}$$

The refracted light has direction

$$\begin{aligned} \mathbf{P} &= (-\cos \phi)\mathbf{N} + (\sin \phi)\mathbf{N}^\perp \\ &= \left(-\sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 \sin^2 \theta}\right)\mathbf{N} + \left(\frac{n_1}{n_2} \sin \phi\right)\mathbf{N}^\perp \\ &= \left(\frac{n_1}{n_2} \cos \theta - \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 \sin^2 \theta}\right)\mathbf{N} + \frac{n_1}{n_2}\mathbf{L} \end{aligned} \quad (20.3)$$

The formula is expressed in terms of the independent input values: the surface normal $\mathbf{N}$, the incoming light direction $\mathbf{L}$, the angle of incidence θ , and the indices of refraction n_1 and n_2 .

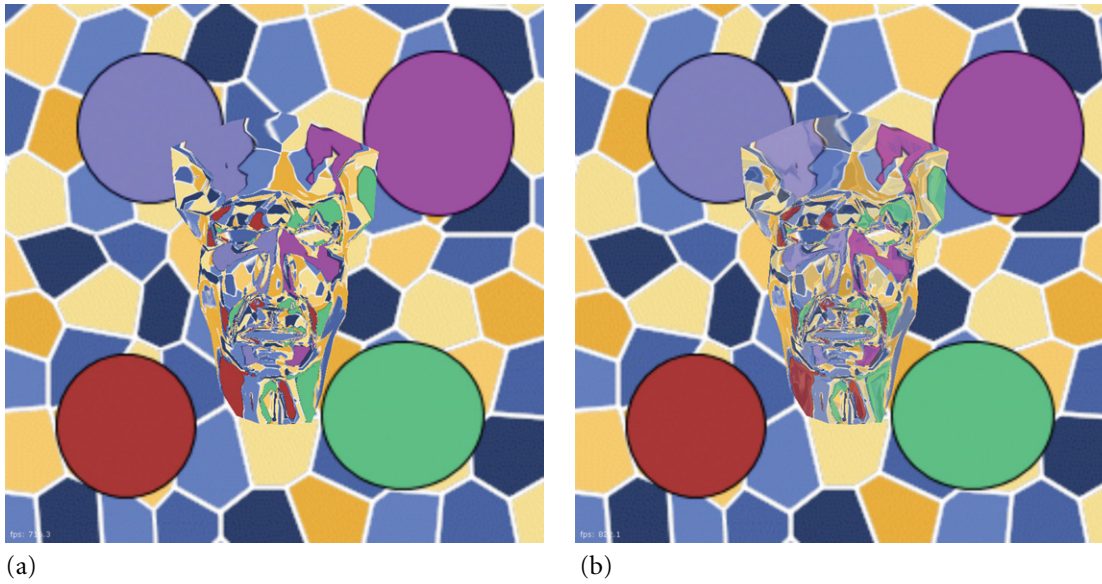


Figure 20.24 Two screen shots from the refraction shader application. (a) Refraction, but no reflection. (b) Refraction and reflection.

Refraction is clearly observed when placing a stick into water. The stick appears to bend at the water surface. Empirical measurements have shown that the index of refraction for air is 1.00029 and the index of refraction for water is 1.333. If the first medium in Figure 20.23 is air and the second medium is water, and if the angle of incidence (angle of reflection) is $\pi/4$ radians (45 degrees), then the refraction angle is $\phi \doteq 0.559328$ radians. This angle is slightly larger than $\pi/6$ radians (30 degrees).

Figure 20.24 shows some screen shots from the sample application *Refraction*. The texture image is a modification of one that is available from the Cg texture library at <http://oss.ckk.chalmers.se/textures/>. The reflection effects in Figure 20.24 (b) are produced by Fresnel reflectance. Notice that rendering using both refraction and reflection appears to be more realistic than with refraction alone; the model has a glassier look to it.

The refraction application uses both a vertex and a pixel shader and calculates refraction and reflection of the environment. The light direction is computed as the view direction from the eye point to a surface vertex. Using the surface normal and the indices of refraction, a refraction vector is computed according to Equation (20.3). This vector is used as a lookup into the environment map. If the environment map is a cube map, a shader can directly perform the lookup. If the environment map is

a sphere map, a few more calculations are needed to obtain the correct texture coordinates. A reflection vector is calculated in a similar manner according to Equation (20.2). A Fresnel factor is calculated to determine the strength of the reflection versus the refraction. For a more realistic refraction, a textured quadrilateral that represents the environment map is placed behind the object.

20.10 PLANAR REFLECTION

An object in a scene can cast reflections onto one or more planar surfaces. Thinking of the planes as mirrors, each plane has some amount of reflectance, say, a value in the interval $[0, 1]$. A reflectance of 0 means the plane does not reflect at all. A reflectance of 1 means the plane fully reflects the image. Values between 0 and 1 give varying degrees of reflectance.

Planar reflections are handled by an Effect-derived class, which makes it a global effect. The class name is `PlanarReflectionEffect` and is in the sample applications folder, `PlanarReflections`. In Wild Magic version 3, planar reflections were implemented by adding a virtual function to the `Renderer` class. Each derived class had to implement that virtual function to handle all the details of drawing the object and its reflections. In Wild Magic version 4, the roles are reversed. The global effect class is responsible for drawing the object and its reflections. By placing the responsibility for drawing complicated effects in the effect class itself, the `Renderer` class does not have to be modified when you add the new effect to the engine.

The class that encapsulates this is `PlanarReflectionEffect` and has the interface

```
class PlanarReflectionEffect : public Effect
{
public:
    PlanarReflectionEffect (int iQuantity);
    virtual ~PlanarReflectionEffect ();

    virtual void Draw (Renderer* pkRenderer, Spatial* pkGlobalObject,
        int iVisibleQuantity, VisibleObject* akVisible);

    // member access
    int GetQuantity () const;
    void SetPlane (int i, TriMeshPtr spkPlane);
    TriMeshPtr GetPlane (int i) const;
    void SetReflectance (int i, float fReflectance);
    float GetReflectance (int i) const;

protected:
    PlanarReflectionEffect ();
```

```

void GetReflectionMatrixAndPlane (int i, Matrix4f& rkReflection,
    Plane3f& rkPlane);

int m_iQuantity;
TriMeshPtr* m_aspkPlane;
float* m_afReflectance;

// Temporary render state for drawing.
AlphaStatePtr m_spkAState;
StencilStatePtr m_spkSState;
ZBufferStatePtr m_spkZState;
};

```

The constructor is passed the number of planes on which the object will cast a reflection. Each plane has an associated reflectance value. The planes and reflectances are all set by the member functions of the class.

This effect is a global effect. You may attach it to a Node object in the scene graph. A specialized drawing function is provided, namely, `PlanarReflectionEffect::Draw`. The abstraction of the drawing routine is

```

enable depth buffering;
enable stencil buffering;

for each reflecting plane do
{
    // See comment (1) after source code.
    disable writing to depth buffer;
    disable writing to frame buffer;
    render plane into stencil buffer;

    // See comment (2) after source code.
    enable writing to depth buffer;
    enable writing to frame buffer;
    render plane to write depth buffer to 'far';
    restore depth buffer state to normal;

    // See comment (3) after source code.
    enable extra clip plane for reflection plane;

    // See comment (4) after source code.
    compute the reflection matrix;
    enable it as a postworld transformation;
}

```

```

// See comment (5) after source code.
reverse the culling direction;

// See comment (6) after source code.
draw node subtree with culling disabled;

restore the cull direction;
disable extra clip plane;
disable the postworld transformation;

// See comment (7) after source code.
enable alpha blending;
set blending color to (r,g,b,a) = (0,0,0,reflectance);
disallow alpha state changes;
render the plane;
allow alpha state changes;
disable alpha blending;
}

disable stencil buffering;
disable depth buffering;
draw the node subtree;

```

The steps of the pseudocode are as follows:

1. Render the reflecting plane into the stencil buffer. No pixels are written to the depth buffer or color buffer, but we use depth buffer testing so that the stencil buffer will not be written where the plane is behind something already in the depth buffer.
2. Render the reflecting plane again by only processing pixels where the stencil buffer contains the plane's assigned stencil value. This time there are no changes to the stencil buffer, and the depth buffer value is reset to the far clipping plane. This is done by setting the range of depth values in the viewport volume to be $[1, 1]$. Since the reflecting plane cannot also be semitransparent, it does not matter what is behind the reflecting plane in the depth buffer. We need to set the depth buffer values to the far plane value (essentially infinity) wherever the plane pixels are, so that when the reflected object is rendered, it can be depth-buffered correctly. Note that the rendering of the reflected object will cause depth values to be written that will appear to be behind the mirror plane. Writes to the color buffer are now enabled. When the reflecting plane is rendered later and blended into the background, which should contain the reflected object, we need to use the same blending function so that the pixels where the reflected object was not rendered will contain the reflecting plane's colors. In that case, the blending result will show

that the reflecting plane is opaque when in reality it was blended with blending coefficients summing to 1.

3. Enable a clip plane so that only objects above the mirror plane are reflected. We do not want objects below the clip plane to be reflected, because we cannot see them.
4. The reflection matrix is computed using a function in the class `Matrix4`.
5. Reverse the cull direction. This is necessary because of the use of the reflection matrix. The triangles in the meshes need to be treated as if their vertices are ordered in the opposite direction that they normally are in. In the actual implementation, a flag is set in the `Renderer` class telling the `CullState` processing code to treat back-facing triangles as front facing, and vice versa. This allows us not to assume that all models use back-facing triangles or all use front-facing triangles.
6. The rendering of the reflected object draws only where the stencil buffer contains the reflecting plane's stencil value. The node object is told to draw itself. The actual function call takes two arguments; the first is the renderer itself and the second is a Boolean flag indicating whether or not to allow culling by bounding volumes. In this case the culling is disabled to allow out-of-view objects to cast reflections.
7. The reflecting plane will be blended with what is already in the frame buffer, either the image of the reflected caster or the reflecting plane. All we want for the reflecting plane at this point is to force the alpha channel to always be the reflectance value for the reflecting plane. The reflecting plane is rendered wherever the stencil buffer is set to the plane's stencil value. The stencil buffer value for the plane will be cleared. The normal depth buffer testing and writing occurs. The frame buffer is written to, but this time the reflecting plane is blended with the values in the frame buffer based on the reflectance value. Note that where the stencil buffer is set, the frame buffer has color values from either the reflecting plane or the reflected object. Blending will use a source coefficient of $1 - \alpha$ for the reflecting plane and a destination coefficient of α for the reflecting plane or reflected object.

The sample application `PlanarReflections` illustrates the planar reflection effect. A biped model is loaded and drawn standing on a plane. Another plane perpendicular to the first is also drawn. The biped casts two reflections, one on the floor plane and one on the wall plane. Figure 20.25 shows a screen shot from the application. The reflectance for the wall mirror is set to be larger than the reflectance of the floor. Press the G key to animate the biped and see the reflections change dynamically. You can also rotate the scene using the virtual trackball. The reflection on the wall will still occur even when the biped itself is out of view of the camera.

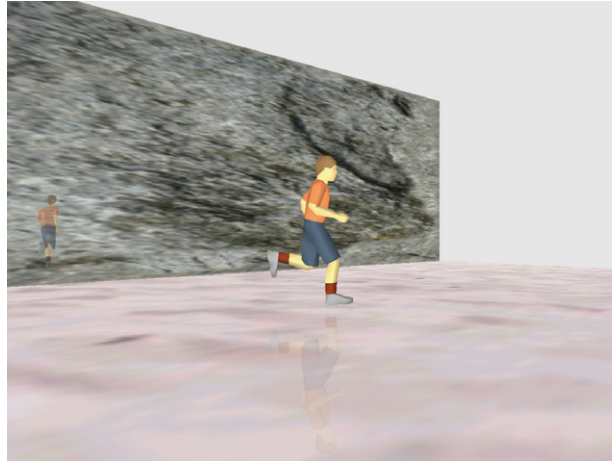


Figure 20.25 Illustration of planar reflections.

20.11 PLANAR SHADOWS

An object in a scene can cast shadows onto one or more planar surfaces. Planar shadows are handled by an Effect-derived class, which makes it a global effect. The class name is `PlanarShadowEffect` and is in the sample applications folder, `PlanarShadows`. In Wild Magic version 3, planar shadows were implemented by adding a virtual function to the `Renderer` class. Each derived class had to implement that virtual function to handle all the details of drawing the object and its shadows. In Wild Magic version 4, the roles are reversed. The global effect class is responsible for drawing the object and its shadows. By placing the responsibility for drawing complicated effects in the effect class itself, the `Renderer` class does not have to be modified when you add the new effect to the engine.

The class that encapsulates this is `PlanarReflectionEffect` and has the interface

```
class PlanarShadowEffect : public Effect
{
public:
    PlanarShadowEffect (int iQuantity);
    virtual ~PlanarShadowEffect ();

    virtual void Draw (Renderer* pkRenderer, Spatial* pkGlobalObject,
        int iVisibleQuantity, VisibleObject* akVisible);
```



```

// member access
int GetQuantity () const;
void SetPlane (int i, TriMeshPtr spkPlane);
TriMeshPtr GetPlane (int i) const;
void SetProjector (int i, LightPtr spkProjector);
LightPtr GetProjector (int i) const;
void SetShadowColor (int i, const ColorRGBA& rkShadowColor);
const ColorRGBA& GetShadowColor (int i) const;

protected:
    PlanarShadowEffect ();

    bool GetProjectionMatrix (int i,
        const BoundingVolume* pkGlobalObjectWorldBound,
        Matrix4f& rkProjection);

    int m_iQuantity;
    TriMeshPtr* m_aspkPlane;
    LightPtr* m_aspkProjector;
    ColorRGBA* m_akShadowColor;

    // temporary render state for drawing
    AlphaStatePtr m_spkAState;
    MaterialStatePtr m_spkMState;
    StencilStatePtr m_spkSState;
    ZBufferStatePtr m_spkZState;
    MaterialEffectPtr m_spkMEffect;
};

```

The constructor is passed the number of planes on which the object will cast a shadow. Each plane has an associated light source for projecting the shadow and a shadow color. The planes, projectors, and colors are all set by the member functions of the class.

This effect is a global effect. You may attach it to a Node object in the scene graph. A specialized drawing function is provided, namely, `PlanarShadowEffect::Draw`. The abstraction of the drawing routine is

```

for each geometric primitive do
{
    draw the primitive;
}

for each projection plane do

```

```

{
    enable depth buffering;
    enable stencil buffering;
    draw the plane; // Stencil keeps track of pixels drawn.

    compute the shadow projection matrix;
    enable it as a postworld transformation;

    enable alpha blending;
    set special material diffuse/alpha to shadow color;

    // drawing controlled only by stencil
    disable depth buffering;
    enable stencil buffering;

    // Shadow occurs only where plane is.
    for each geometric primitive do
    {
        save material of primitive;
        save effects of primitive;
        attach special material to primitive;
        draw primitive using material-only shader;
        detach special material from primitive;
        restore effects of primitive;
        restore material of primitive;
    }

    disable stencil buffering;
    disable alpha blending;

    disable the postworld transformation;
}

```

The shadow caster is drawn first. Each projection plane is processed one at a time. The plane is drawn with the stencil buffer enabled so that you keep track of those pixels affected by the plane. The shadow caster needs to be drawn into the plane from the perspective of the light projector. This involves computing a shadow projection matrix, which I will discuss in a moment. The matrix is pushed onto the model view matrix stack so that the resulting transformation places the camera in the correct location and orientation to render the caster onto the projection plane, which acts as the view plane. The only colors we want for the rendering are the shadow colors. This is accomplished by drawing each geometric primitive using a material whose diffuse and alpha channels are set to the shadow color. The caster is rendered into the plane,

but with stencil buffering enabled yet again, the only pixels blended with the shadow color are those that were drawn with the plane. The stencil buffer only allows those pixels to be touched. The remainder of the code is the restoration of rendering state to what it was before the function call was made.

The projection matrix construction requires a small amount of mathematics. The projection plane is implicit in the `TriMesh` representation of the plane. We need to know the equation of the plane in world coordinates. The `TriMesh::GetWorldTriangle` function call returns the three vertices of the first triangle in the mesh of the plane, and from these vertices we can construct the plane equation. The light source is either directional, in which case the projection is an oblique one, or positional, in which case the projection is a perspective one. The homogeneous matrices for these projections are computed by the `Matrix4` class, in particular by the function `MakeObliqueProjection` or `MakePerspectiveProjection`.

The sample application `PlanarShadows` illustrates the projected, planar shadow effect. A biped model is loaded and drawn standing on a plane. Another plane perpendicular to the first is also drawn. The biped casts two shadows, one on the floor plane and one on the wall plane. Figure 20.26 shows a screen shot from the application. The light is a point source. You can move the location with the x, y, and z keys (both lowercase and uppercase). Press the G key to animate the biped and see the shadow change dynamically.



Figure 20.26 Illustration of projected, planar shadows.

20.12 PROJECTED TEXTURES

A *projected texture* is a texture that is applied to an object as if it were projected from a light source onto the object. You may think of this as the model of a motion picture projection system that casts an image onto a screen. Another realistic example is light passing through a stained-glass window and tinting the objects in a room. The class that encapsulates this effect is `ProjectedTextureEffect`:

```
class ProjectedTextureEffect : public ShaderEffect
{
public:
    ProjectedTextureEffect (Camera* pkProjector,
                           const char* acProjectorImage,
                           Light* pkLight);
    virtual ~ProjectedTextureEffect ();

    virtual void SetGlobalState (int iPass, Renderer* pkRenderer,
                                bool bPrimaryEffect);
    virtual void RestoreGlobalState (int iPass, Renderer* pkRenderer,
                                    bool bPrimaryEffect);

protected:
    // streaming
    ProjectedTextureEffect ();

    CameraPtr m_spkProjector;
    LightPtr m_spkLight;
};
```

The first input to the constructor is the projector, which is a `Camera` object to allow perspective projection into a view frustum. The projector actually represents the view of the world from the light source. The second input is the image to be projected. The third input is an actual light and must be directional for the purposes of the sample application, `ProjectedTextures`, which blends the projected texture and the directional lighting together.

Because dynamic lighting is used, the renderer needs to be informed about the light. Moreover, it needs to know about the projector. Both parameters are made known through the `SetGlobalState` call. The function `RestoreGlobalState` resets the renderer state after drawing has occurred. This pair of functions was discussed in Sections 3.3.3 and 3.4.2.

The vertex program is

```

void v_ProjectedImage
(
    in float4      kModelPosition : POSITION,
    in float3      kModelNormal : NORMAL,
    out float4     kClipPosition : POSITION,
    out float4     kVertexColor : COLOR,
    out float4     kTCoord : TEXCOORD0,
    uniform float4x4 WVPMatrix,
    uniform float4x4 ProjectorMatrix,
    uniform float3  CameraModelPosition,
    uniform float3  MaterialEmissive,
    uniform float3  MaterialAmbient,
    uniform float4  MaterialDiffuse,
    uniform float4  MaterialSpecular,
    uniform float3  Light0ModelDirection,
    uniform float3  Light0Ambient,
    uniform float3  Light0Diffuse,
    uniform float3  Light0Specular)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    // Compute directional lighting.
    float fDiff, fSpec;
    GetDirectionalLightFactors((float3)kModelPosition,kModelNormal,
        CameraModelPosition,Light0ModelDirection,MaterialSpecular.a,
        fDiff,fSpec);

    float3 kColor = MaterialAmbient*Light0Ambient;
    if (fDiff > 0.0f)
    {
        kColor += fDiff*MaterialDiffuse.rgb*Light0Diffuse;
        if (fSpec > 0.0f)
        {
            kColor += fSpec*MaterialSpecular.rgb*Light0Specular;
        }
    }

    kVertexColor.rgb = MaterialEmissive + kColor;
    kVertexColor.a = MaterialDiffuse.a;

    // Compute the projected texture coordinates.
    kTCoord = mul(kModelPosition,ProjectorMatrix);
}

```

The clip-space position is computed by transforming the model-space position. The remainder of the code except for the last line is the same code that occurs in `Lighting.cg`, vertex program `L1d`, for directional lighting. This requires the vertices to have normals specified. The vertex colors are output from the program. The last line of code computes the texture coordinates for the projected texture. The uniform constant `ProjectorMatrix` is one of the special constants associated with the renderer. The renderer is told about the projector during the function call `ProjectedTextureEffect::SetGlobalState`. When the uniform constants are set up for a call to the vertex program, the projection matrix associated with the projector is automatically computed and passed to the graphics API. See the function `Renderer::SetConstantProjectorMatrix` in the graphics engine.

The pixel program is

```
void p_ProjectorTexture
(
    in float4      kVertexColor : COLOR,
    in float4      kTCoord   : TEXCOORD0,
    out float4     kPixelColor : COLOR,
    uniform sampler2D ProjectorSampler)
{
    float4 kProjectorColor = tex2Dproj(ProjectorSampler,kTCoord);
    kPixelColor = kProjectorColor*kVertexColor;
}
```

Cg has a sampler for looking up the colors in a projected texture. The texture coordinate is of the form (s, t, r, q) . The sampler computes $(s', t') = (s/q, t/q)$ and uses this as a lookup into the projected texture image. The Wild Magic software renderer has an implementation of a projected texture sampler, class `SoftSamplerProj`.

The sample application `ProjectedTextures` illustrates the projected texture effect. A mesh representing a face is loaded and has no texture associated with it. A sunfire texture is projected onto the face. Figure 20.27 shows the face in a couple of orientations. Notice that the face moves relative to the observer, but the projected texture does not since the projector is fixed in space in this application.

20.13 SHADOW MAPS

Casting shadows is a desirable effect in many applications. One of the simplest methods, planar shadows, was discussed in Section 20.11. The method requires that the recipient of the shadow be a plane. It would be nice, though, to cast shadows onto nonplanar objects. One method to do this is using *shadow maps*, where a light is used to cast the shadows.

The process requires two rendering passes. The scene is rendered from the light's point of view. All that matters in this pass is computing the distances from the light

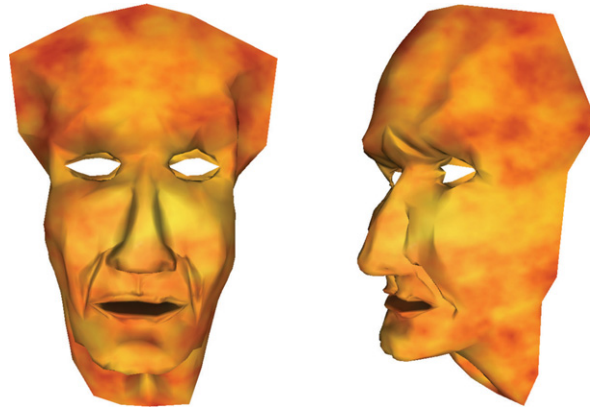


Figure 20.27 Illustration of projected textures.

source to the geometric primitives. These are stored as pixel depths and stored in a *depth texture* (the shadow map). The scene is then rendered from the camera's point of view. The depth texture is used as a projected texture, projected from the light's point of view. At each pixel, the depth texture value is compared to the fragment's distance from the light source. If the depth texture value is greater than the distance, the pixel is in shadow—another fragment is closer to the light and casts a shadow on this pixel.

Although this sounds simple, there are quite a few setup details to take care of.

- An offscreen buffer must be created to store the depth texture, and the renderer must be told to compute only the depths when rendering the scene from the light's point of view. Getting an offscreen buffer system to work has been the source of great pain for many newcomers to graphics. OpenGL's pbuffer construct is problematic because the details of construction are specific to the platform (Windows Wgl, Apple's Agl, Unix/Linux Glx). The recent introduction of framebuffer objects has made this easier to do since the construction is independent of platform.
- The texture resource management system must support depth textures. These are handled slightly differently from regular textures.
- Precision problems can cause the depth texture and other textures to have z-fighting. To avoid this, a polygon offset may be used, but the bias parameters are specific to the objects, the camera model, and the environment they live in. A better option would be to have access to the samplers, configuring them to sample differently, but that capability is not yet available on GPUs.
- Ideally, the depth texture may be rendered to as a texture, so it may be applied as a texture (without first having to copy it from video memory to system memory),

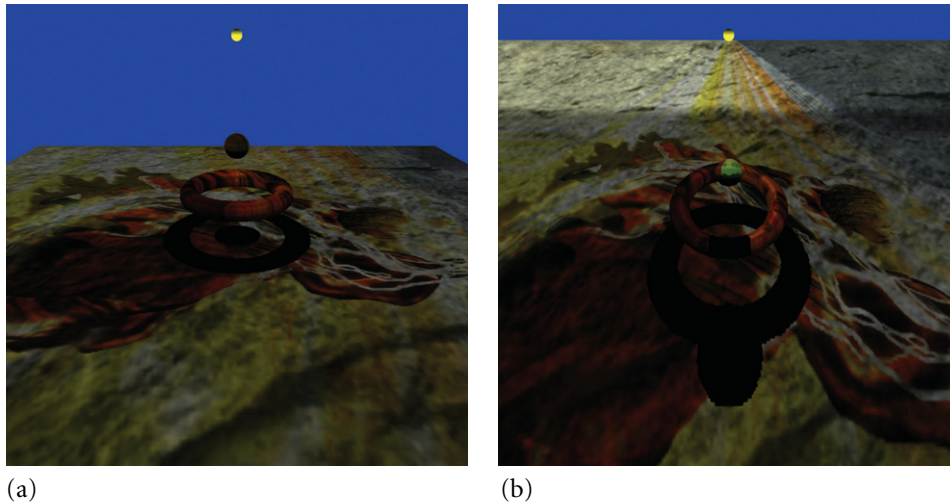


Figure 20.28 Two screen captures from the ShadowMaps sample.

copied to another texture object, and then sent back to video memory. Render-to-texture is becoming more common, but older hardware and drivers might not support this.

The sample application to illustrate this concept is ShadowMaps. Figure 20.28 shows two screen captures. The light is represented by the yellow globe. The larger sphere and the torus have material colors and are lit but also allow a projected texture to be blended with their colors. The ground plane has a base texture (stone) and accepts the projected texture and the shadow map. Figure 20.28 (b) shows the scene slightly tilted, so the shadows are cast closer to the observer.

20.14 VOLUMETRIC FOG

Standard depth-based fog is useful to hide the abrupt change that would otherwise occur at the far plane. There are times, though, when you want a fog effect such that fog occurs in a localized region *anywhere* you choose in the view frustum. This could very well be in a region close to the near plane. A mechanism for creating such effects is *volumetric fog*.

The idea is to generate per-vertex fog factors and use these to blend the fog color with the other vertex attributes. To obtain a localized effect, you add nondisplayable objects to your scene. Such an object is called a *fog generator*. For each vertex, a ray is cast from the eye position to the vertex position. If the ray does not intersect any

of the fog generators, then the fog factor is 1 and the vertex color is whatever the vertex attributes generate. If the ray does intersect a fog generator, then a fog factor is chosen to be smaller than 1 and proportional to the size of the intersection of the ray with that generator. The intersection calculations are performed on the CPU, so to keep computational time to a minimum, the fog generators are usually chosen to be simple convex objects; for example, a sphere, box, or slab are reasonable choices. For convex fog generators, the intersection set of the ray and generator is a single line segment, a single point, or empty.

The per-vertex fog factors can be computed using a vertex program. The sample application `VolumeFog` illustrates how to do this. The geometric primitive is a height field. A gray-scale texture is used to define the height field. Once the heights are obtained (from the red channel), the texture image is modified to produce shades of green (small height) and red (large height). Vertex colors are used to store the fog color and the fog factor. This allows you to actually assign different fog colors to different vertices. In the sample source code, the fog colors are all set to white. The alpha channel of the vertex colors is used to store the fog factor.

A rectangular slab parallel to the ground plane is used as the fog generator. As the camera is moved around the terrain, a function `UpdateFog` is called. Its job is to compute the intersections of the rays (from eye position to vertex position) and the rectangular slab. The rays are of the form $\mathbf{E} + t\mathbf{P}$, where $\mathbf{E}$ is the eye position and $\mathbf{P}$ is a vertex position. All we care about are those points for which $t \in [0, 1]$. The intersection of the ray with the slab produces a subinterval $[t_0, t_1]$. The length L of the interval is used to generate the fog factor $f = L/(L + L_0)$, where L_0 is a user-defined constant. The application chooses $L_0 = 8$, so f can never be 1, which prevents oversaturation by the fog color. `UpdateFog` handles the cases when $\mathbf{E}$ is below, within, and above the slab.

The shader programs are

```
void v_VolumeFog
(
    in float4      kModelPosition : POSITION,
    in float4      kModelColor : COLOR,
    in float2      kModelTCoord : TEXCOORD0,
    out float4     kClipPosition : POSITION,
    out float4     kVertexColor : COLOR,
    out float2     kTCoord : TEXCOORD0,
    uniform float4x4 WVPMatrix)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    // Pass through the vertex color and texture coordinate.
    kVertexColor = kModelColor;
    kTCoord = kModelTCoord;
}
```

```

void p_VolumeFog
(
    in float4      kVertexColor : COLOR,
    in float2      kTCoord : TEXCOORD0,
    out float4     kPixelColor : COLOR,
    uniform sampler2D BaseSampler)
{
    // The blending equation is
    // (rf,gf,bf) = (1 - av) * (rt,gt,bt) + av * (rv,gv,bv),
    // where (rf,gf,bf) is the final color, (rt,gt,bt)
    // is the texture color, and (rv,gv,bv,av) is the
    // vertex color.

    float4 kTextureColor = tex2D(BaseSampler,kTCoord);
    kPixelColor.rgb = (1.0f - kVertexColor.a)*kTextureColor.rgb +
        kVertexColor.a*kVertexColor.rgb;
    kPixelColor.a = 1.0f;
}

```

The blending uses the fog factor, stored in the alpha channel of the incoming color, and applies a linear interpolation to the RGB channels of the incoming color and of the texture color.

Figure 20.29 shows screen captures from the sample application. Some artifacts are noticeable on the distant mountains. This is due to the coarse-level tessellation of the height field. You can always switch to per-pixel fog, but this can be very expensive

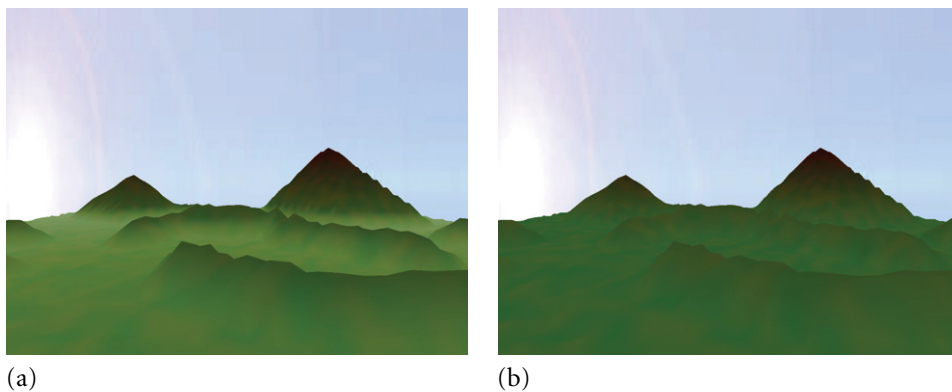


Figure 20.29 (a) The height field rendered with volumetric fog. (b) The height field rendered without volumetric fog.

if you use function calls in the pixel program. For finer control using per-pixel fog, consider using a texture approach.

20.15 SKINNING

I had discussed skin-and-bones animation in Section 5.4. The computation of the world coordinates for the vertex positions is handled by the `SkinController` class, all done on the CPU. The GPU can also handle skin-and-bones animation.

The sample application `Skinning` illustrates skinning with four matrices. The vertex program is

```
void v_Skinning
(
    in float4      kModelPosition : POSITION,
    in float3      kModelColor : COLOR,
    in float4      kWeight : TEXCOORD0,
    out float4     kClipPosition : POSITION,
    out float3     kVertexColor : COLOR,
    uniform float4x4 WVPMatrix,
    uniform float4x4 SkinningMatrix0,
    uniform float4x4 SkinningMatrix1,
    uniform float4x4 SkinningMatrix2,
    uniform float4x4 SkinningMatrix3)
{
    // Calculate the position by adding together a convex combination of
    // transformed positions.
    float4 kSkinPos0 = mul(kModelPosition, SkinningMatrix0) * kWeight.x;
    float4 kSkinPos1 = mul(kModelPosition, SkinningMatrix1) * kWeight.y;
    float4 kSkinPos2 = mul(kModelPosition, SkinningMatrix2) * kWeight.z;
    float4 kSkinPos3 = mul(kModelPosition, SkinningMatrix3) * kWeight.w;
    float4 kSkinPosition = kSkinPos0 + kSkinPos1 + kSkinPos2 + kSkinPos3;

    // Transform the position from model space to clip space.
    kClipPosition = mul(kSkinPosition, WVPMatrix);

    // Pass through the color.
    kVertexColor = kModelColor;
}
```

This is just one way to do skinning in a shader—by passing in four (or more) skinning matrices to transform the vertices. Another way is to pass in an arbitrary number of skinning matrices as constants, and then use the texture coordinates as indices into the global array of constants to select the correct skinning matrix. If you

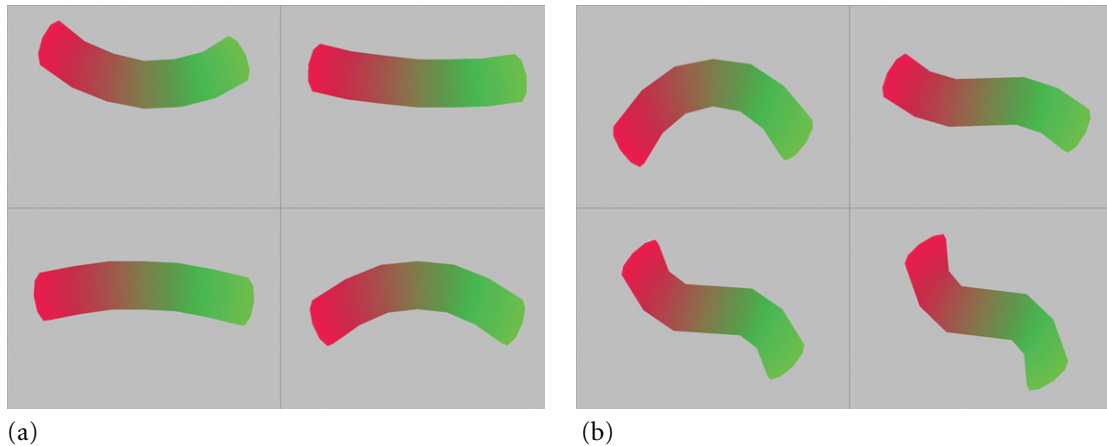


Figure 20.30 Two screen shots from the skinning application. The bones are randomly generated to cause the object to continuously deform. The sequence of deformations is (a) from left to right, then (b) from top to bottom.

only have a small number of skinning matrices, the method shown here is the easiest way to handle skinning.

In this simple application, the vertices are assigned only vertex colors. Figure 20.30 shows a few screen captures from the application.

20.16 IRIDESCENCE

Another optical effect related to reflection and refraction at the surface between two media, *iridescence* is caused by interference when light is partially transmitted through the surface and partially reflected by the surface. The classical occurrence of iridescence is with soap bubbles obtained by dipping a circular wire into a soapy solution. When the wire is removed from the solution, bands of color will appear on the soap film. The physical mechanism is a bit complex, the interference caused by some of the light waves being reflected out of phase and interacting with other light waves in phase. The typical hack by computer graphics programmers is to simulate the effects with view-dependent coloring.

The *Iridescence* shader calculates a per-pixel viewing direction and normal vector. In the pixel shader, a dot product of the viewing direction and normal is computed and used as an input to a 1D gradient texture lookup. When viewed straight on, the gradient texture has a green tint. When viewed at an angle, the tint is blue.

The lookup texture color is blended with the original texture and produces an iridescent sheen.

The vertex program is

```
void v_Iridescence
(
    in float4      kModelPosition : POSITION,
    in float3      kModelNormal : NORMAL,
    in float2      kInBaseTCoord : TEXCOORD0,
    out float4     kClipPosition : POSITION,
    out float2     kOutBaseTCoord : TEXCOORD0,
    out float      fOutInterpolateFactor : TEXCOORD1,
    out float3     kWorldNormal : TEXCOORD2,
    out float3     kEyeDirection : TEXCOORD3,
    uniform float4x4 WVPMatrix,
    uniform float4x4 WMatrix,
    uniform float3  CameraWorldPosition,
    uniform float   InterpolateFactor)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    // Transform the position from model space to world space.
    float3 kWorldPosition = mul(kModelPosition,WMatrix).xyz;

    // Transform the normal from model space to world space.
    // In case the model-to-world matrix has nonunit scales,
    // the resulting vector must be normalized. Map the
    // vector to [0,1]^3.
    kWorldNormal = MapToUnit(normalize(mul(kModelNormal,
        (float3x3)WMatrix)));

    // Calculate the eye direction. Map the vector to [0,1]^3.
    kEyeDirection = MapToUnit(
        normalize(kWorldPosition-CameraWorldPosition));

    // Pass through the base texture coordinate.
    kOutBaseTCoord = kInBaseTCoord;

    // Pass through the interpolation factor.
    fOutInterpolateFactor = InterpolateFactor;
}
```

The object to which the shader is attached has a 2D base texture. The texture colors are blended with the 1D gradient mentioned previously. The blending uses linear interpolation with `InterpolateFactor` used for the weights. The application can modify this to see what effect it has.

The pixel program is

```
void p_Iridescence
(
    in float2      kBaseTCoord : TEXCOORD0,
    in float       fInterpolateFactor : TEXCOORD1,
    in float3      kWorldNormal : TEXCOORD2,
    in float3      kEyeDirection : TEXCOORD3,
    out float4     kPixelColor : COLOR,
    uniform sampler2D BaseSampler,
    uniform sampler1D GradientSampler)
{
    // Map the vectors to [-1,1]^3.
    kWorldNormal = MapFromUnit(kWorldNormal);
    kEyeDirection = MapFromUnit(kEyeDirection);

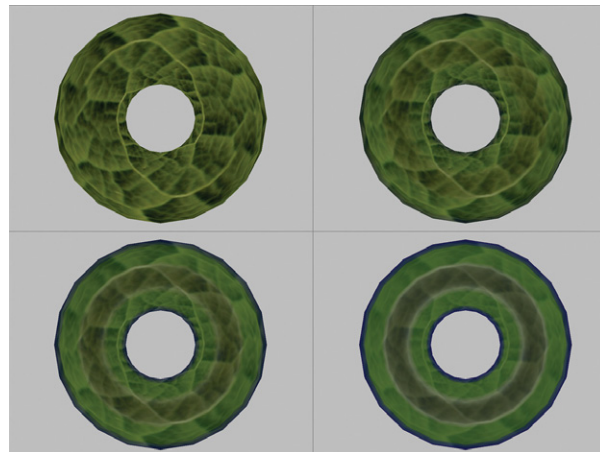
    // Calculate a Fresnel factor for a view-dependent lookup
    // into a gradient texture. A different color/saturation
    // occurs depending on the angle used for viewing.
    float fFresnel = 1 + dot(kWorldNormal,kEyeDirection);
    fFresnel = fFresnel*fFresnel;

    float3 kBaseColor = tex2D(BaseSampler,kBaseTCoord).xyz;

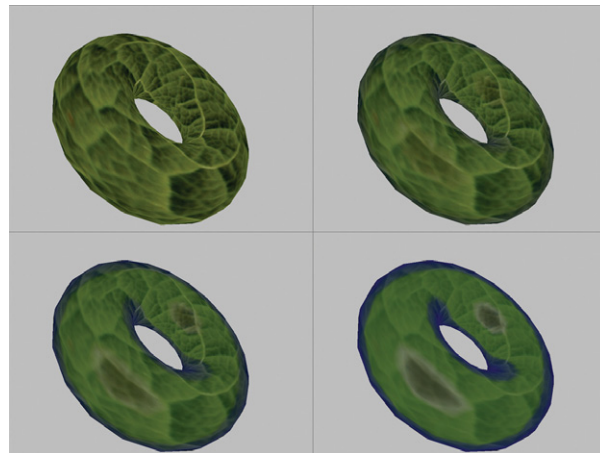
    // The small perturbation of the Fresnel factor eliminates
    // some spotting where values are nearly zero.
    float fGradientTCoord = saturate(fFresnel + 1.0f/256.0f);
    float3 kGradientColor = tex1D(GradientSampler,fGradientTCoord).xyz;

    // Blend the colors for the pixel color.
    kPixelColor.rgb = lerp(kBaseColor,kGradientColor,
        fInterpolateFactor);
    kPixelColor.a = 1.0f;
}
```

Figure 20.31 shows some screen shots from the sample application *Iridescence*. The leaf texture image is available from the Cg texture library at <http://oss.ckk.chalmers.se/textures/>. The upper-left quadrants of both images show the torii with no iridescence. The interpolation factor used to control the iridescence is set to zero in



(a)



(b)

Figure 20.31 Screen shots from the iridescence shader application. The two sets of images show a textured torus in two different orientations and with various amounts of interpolation to produce the iridescent sheen.

the application. The upper-right quadrants show the torii using an interpolation factor of 0.3. The lower-left quadrants use an interpolation factor of 0.5, the renderings having good quality texture detail and iridescence. The lower-right quadrants use an interpolation factor of 0.7. The iridescence is stronger but at the cost of quality in the texture detail.

20.17 WATER EFFECTS

A shader program that is more complex than the previously mentioned ones shows a rippling ocean water effect with large waves and small bump-mapped ripples. This shader is built using ideas from the book [Eng02], in particular the article, “Rendering Ocean Water,” by John Isidoro, Alex Vlachos, and Chris Brennan. The sky texture is edited from Jeremy Engleman’s page of public textures http://art.net/~jeremy/photo/public_texture/. The plasma height map was made with the GNU Image Manipulation Program (GIMP), using the render sky plasma2 effect with the “tile horizontally” and “tile vertically” options enabled. The GIMP is available at www.gimp.org.

The `RipplingOcean` application has both a vertex shader and a pixel shader. The vertex shader is responsible for creating the wave displacement effect. The pixel shader is responsible for calculating the water color, the diffuse lighting, and the specular reflection, and for putting it all together.

The vertex shader is responsible for many aspects of the rendering. First, the wave effect is obtained by creating a surface that is the sum of four sinusoidal waves, each propagating in the 2D tangent space of the surface. The waves have a specific height along the normal direction, as well as a speed, direction, and offset in that direction. Using only one or two sinusoidal components give the water an unrealistic appearance. Four sinusoidal waves gives it a very undulating effect. You can easily add another four waves if you want yet more control over fine-scale variations.

Second, the vertex shader also calculates a number of vectors that the pixel shader uses. A new tangent vector, normal vector, and binormal vector are computed, all based on the cosines of the wave. These vectors are used by the pixel shader to generate a coordinate frame at each pixel. The vertex shader also calculates a view vector for the pixel shader. Finally, it creates two texture coordinates. These coordinates vary with different wave speeds, with one coordinate inverted relative to the other, so that the two image textures are never aligned, a condition that leads to an unnatural rendering.

The pixel shader first samples the plasma height map, which has been converted into a normal map, with both texture coordinates. The resulting bump maps are averaged together to form a new normal vector for the current pixel. Using the normal, tangent, and binormal vectors, the bump map value is transformed into world space. This value becomes the new normal for the pixel, thereby causing the ripple effect.

The water color is calculated by computing the dot product of the originally calculated normal vector and the view direction and using it as an index into a lookup table for a gradient. When the view direction is nearly perpendicular to the water surface, the water has a green tint. When the view direction is nearly parallel, for example, when looking at the water in the distance, the water has a blue tint. The originally calculated normals are used rather than the new normals, because the latter

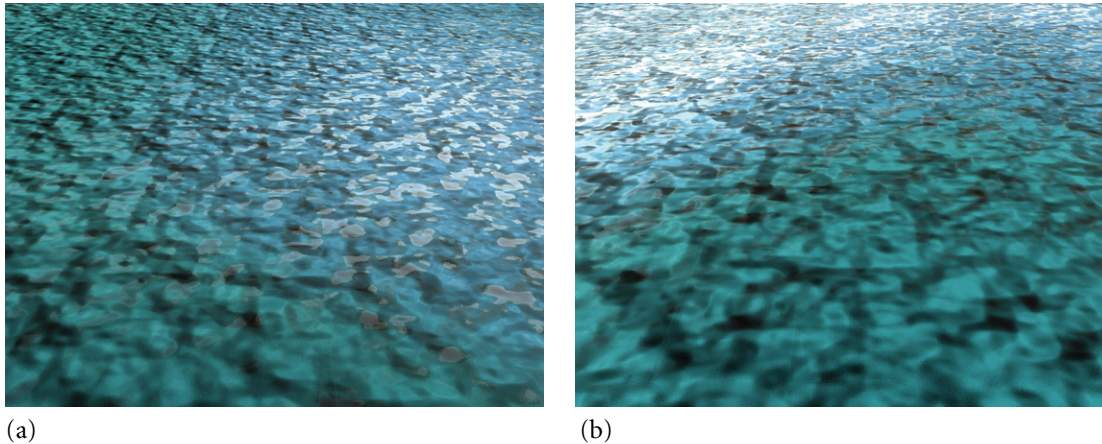


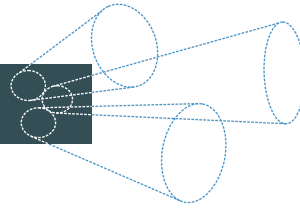
Figure 20.32 Two screen shots from the rippling ocean application. The images were captured at two different times in the simulation.

vector field is too high-frequency for the colors to look realistic. Because of the bump-mapping, blue and green patches appear to be equally distributed over the entire water surface.

The diffuse color is calculated as a dot product of the normal with a directional light. A specular reflection is also calculated. The view direction is reflected through the new normal and a color is looked up from the background image. The magnitude of this color is squared to emphasize the bright parts and then multiplied by the background color. The resulting specular color is multiplied by a Fresnel factor as a dot product of the view direction and the normal. The visual effect is that the water has a large reflectance when the view direction is nearly perpendicular to the surface normal. The reflectance is small when the view direction is nearly parallel to the surface normal. All calculated colors are then combined to obtain the final water color.

Figure 20.32 shows some screen shots from the `RipplingOcean` application. The `RipplingOcean` shader required many tweaks to make the water look realistic. The most important contributor to the realism is the specular reflection. Without it, the shadowing from the bump-mapping looks very strange. In addition to the specular reflection, offsetting the amount of shadowing from the bump-mapping required that a certain amount of ambience be added to the diffuse color term. Adjusting the ambient color gives the water an appearance anywhere from that seen at sunset to full noon on a clear day. The application has various controls to adjust at run time, including adjusting the wave height, the wave speed, the ripple frequency, the ripple texture coordinate repeat factor, and the addition of ambient lighting.

APPENDIX



CREATING A SHADER IN WILD MAGIC

There are two basic ways to create a shader effect in Wild Magic version 4. The first method is *classless* in that you assemble all the components within your application code and attach the effect to the geometric primitive of interest. The second method involves creating a new class, either derived from `ShaderEffect` for a local effect or derived from `Effect` for a global effect that requires its own drawing function.

The sections in this appendix illustrate the two methods for a particular effect. The sample application is located in

`GeometricTools/WildMagic4/SampleGraphics/BlendedTerrain`

and contains a height field representing a small piece of terrain. The terrain will receive a blend of two 2D textures, one representing grass and the other representing stone. The blending factor will be proportional to the height. The smaller the height, the more grass will appear in the blend. The larger the height, the more stone will appear in the blend. Rather than rely on the height values at the vertices, a 1D texture is used to represent the height. This gives you some additional control over the blending factor via the 1D texture's image values instead of relying solely on vertex height. Moreover, the blending factors allow for a nonlinear adjustment to help control how much grass and how much stone are blended together.

A cloud layer will cast cloud shadows on the terrain, but the clouds themselves are not rendered geometry. A sky dome is used for rendering the sky. The clouds are represented by a 2D texture. This texture will be orthogonally projected onto

the terrain. However, the cloud shadows will move based on a direction of motion specified by the application.

A.1 SHADER PROGRAMS FOR AN ILLUSTRATIVE APPLICATION

The Cg vertex program is

```
void v_BlendedTerrain
(
    in float4      kModelPosition : POSITION,
    in float2      kInGroundTCoord : TEXCOORD0,
    in float       kInBlendTCoord : TEXCOORD1,
    in float2      kInCloudTCoord : TEXCOORD2,
    out float4     kClipPosition : POSITION,
    out float2     kOutGroundTCoord : TEXCOORD0,
    out float      kOutBlendTCoord : TEXCOORD1,
    out float2     kOutCloudTCoord : TEXCOORD2,
    out float2     kOutFlowDirection : TEXCOORD3,
    uniform float4x4 WVPMatrix,
    uniform float2  FlowDirection)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    // Pass through the texture coordinates.
    kOutGroundTCoord = kInGroundTCoord;
    kOutBlendTCoord = kInBlendTCoord;
    kOutCloudTCoord = kInCloudTCoord;

    // Pass through the flow direction, to be used as an offset
    // in the pixel program.
    kOutFlowDirection = FlowDirection;
}
```

The Cg pixel program is

```
void p_BlendedTerrain
(
    in float2      kGroundTCoord : TEXCOORD0,
    in float       kBlendTCoord : TEXCOORD1,
    in float2      kCloudTCoord : TEXCOORD2,
    in float2      kFlowDirection : TEXCOORD3,
```

```

out float4      kPixelColor : COLOR,
uniform float    PowerFactor,
uniform sampler2D GrassSampler,
uniform sampler2D StoneSampler,
uniform sampler1D BlendSampler,
uniform sampler2D CloudSampler)
{
    float4 kGrassColor = tex2D(GrassSampler,kGroundTCoord);
    float4 kStoneColor = tex2D(StoneSampler,kGroundTCoord);
    float4 kBlendColor = tex1D(BlendSampler,kBlendTCoord);

    float2 kOffsetCloudTCoord = kCloudTCoord + kFlowDirection;
    float4 kCloudColor = tex2D(CloudSampler,kOffsetCloudTCoord);

    float fStoneWeight = pow(kBlendColor.r,PowerFactor);
    float fGrassWeight = 1.0f - fStoneWeight;
    kPixelColor = kCloudColor*(fGrassWeight*kGrassColor +
        fStoneWeight*kStoneColor);
}

```

The Cg programs can be compiled using NVIDIA's cgc command-line compiler. The batch files in the GeometricTools/WildMagic4/Bin directory may be used by executing them in the directory containing the cg file.

```

GLVProgram BlendedTerrain BlendedTerrain
GLPProgram BlendedTerrain BlendedTerrain
DXVProgram BlendedTerrain BlendedTerrain
DXPPProgram BlendedTerrain BlendedTerrain

```

The prefix GL is for OpenGL and the prefix DX is for DirectX. The output files are

```

v_BlendedTerrain ogl.wmsp
p_BlendedTerrain ogl.wmsp
v_BlendedTerrain dx9.wmsp
p_BlendedTerrain dx9.wmsp

```

respectively.

The “compiled” shader programs for the software renderer must be (for now) manually written. It is simple enough to write these based on the Cg code itself. I copy the dx9.wmsp files and change the extensions on the copies to be sft.wmsp. I then format the comments in the first part of the files to conform to what the Program parser expects. Look at other samples to see what this format is; for example, look at the Iridescence software shader programs. The vertex program is in file v_BlendedTerrain.sft.wmsp:

```

// software vertex shader generated by Wild Magic
//
// var float4    kModelPosition    $vin.POSITION
// var float2    kInGroundTCoord   $vin.TEXCOORD0
// var float     kInBlendTCoord    $vin.TEXCOORD1
// var float2    kInCloudTCoord    $vin.TEXCOORD2
// var float4    kClipPosition     $vout.POSITION
// var float2    kOutGroundTCoord   $vout.TEXCOORD0
// var float     kOutBlendTCoord    $vout.TEXCOORD1
// var float2    kOutCloudTCoord   $vout.TEXCOORD2
// var float2    kOutFlowDirection $vout.TEXCOORD3
// var float4x4  WVPMatrix         c[0]
// var float2    FlowDirection     c[4]

#include "Wm4SoftRenderer.h"
#include "Wm4Matrix4.h"
#include "Wm4Vector3.h"

namespace Wm4
{
void v_BlendedTerrain (const float* afRegister, const float* afInVertex,
                      float* afOutVertex)
{
    // Get the register items.
    const Matrix4f& rkWVPMatrix = *(const Matrix4f*)&afRegister[0];
    const Vector2f& rkFlowDirection = *(const Vector2f*)&afRegister[16];

    // Get the input items.
    Vector4f kModelPosition(afInVertex[0],afInVertex[1],afInVertex[2],1.0f);
    const Vector2f& rkInGroundTCoord = *(const Vector2f*)&afInVertex[3];
    float fInBlendTCoord = afInVertex[5];
    const Vector2f& rkInCloudTCoord = *(const Vector2f*)&afInVertex[6];

    // Access the output items.
    Vector4f& rkClipPosition = *(Vector4f*)&afOutVertex[0];
    Vector2f& rkOutGroundTCoord = *(Vector2f*)&afOutVertex[4];
    float& rfOutBlendTCoord = afOutVertex[6];
    Vector2f& rkOutCloudTCoord = *(Vector2f*)&afOutVertex[7];
    Vector2f& rkOutFlowDirection = *(Vector2f*)&afOutVertex[9];

    // *** program ***

```

```

// Transform the position from model space to clip space.
rkClipPosition = kModelPosition*rkWVPMatrix;

// Pass through the texture coordinates.
rkOutGroundTCoord = rkInGroundTCoord;
rfOutBlendTCoord = fInBlendTCoord;
rkOutCloudTCoord = rkInCloudTCoord;

// Pass through the flow direction, to be used as an offset
// in the pixel program.
rkOutFlowDirection = rkFlowDirection;
}

WM4_IMPLEMENT_VPROGRAM(BlendedTerrain);
WM4_REGISTER_VPROGRAM(BlendedTerrain);
}

```

The majority of the work is converting the Cg code to C++ code. The register constants are set up by accessing the input array `afRegister`. The indices into this array are multiples of 4 since each register is a 4-float quantity. The input `FlowDirection` is a 2-float quantity, so only the first two float values of the 4-float register are used.

The macros used at the end of the code block are necessary for the vertex program to be registered with the software renderer. The registration guarantees that the software renderer can locate the vertex program during application execution.

The pixel program is in file `p_BlendedTerrain.sft.wmsp`:

```

// software pixel shader generated by Wild Magic
//
// var float2    kGroundTCoord    $vin.TEXCOORD0
// var float     kBlendTCoord     $vin.TEXCOORD1
// var float2    kCloudTCoord     $vin.TEXCOORD2
// var float2    kFlowDirection   $vin.TEXCOORD3
// var float4    kPixelColor       $vout.COLOR
// var float     PowerFactor       c[1]
// var sampler2D GrassSampler      texunit 0
// var sampler2D StoneSampler      texunit 1
// var sampler1D BlendSampler      texunit 2
// var sampler2D CloudSampler      texunit 3

#include "Wm4SoftRenderer.h"
#include "Wm4ColorRGBA.h"
#include "Wm4Vector3.h"

```

```

namespace Wm4
{
    ColorRGBA p_BlendedTerrain (const float* afRegister,
                               SoftSampler** apkSampler, const float* afInPixel)
    {
        // Get register values.
        float fPowerFactor = afRegister[0];

        // Get samplers.
        SoftSampler& rkGrassSampler = *apkSampler[0];
        SoftSampler& rkStoneSampler = *apkSampler[1];
        SoftSampler& rkBlendSampler = *apkSampler[2];
        SoftSampler& rkCloudSampler = *apkSampler[3];

        // Get input values.
        const float* afGroundTCoord = &afInPixel[0];
        const float* afBlendTCoord = &afInPixel[2];
        const float* afCloudTCoord = &afInPixel[3];
        const float* afFlowDirection = &afInPixel[5];

        // *** program ***

        ColorRGBA kGrassColor = rkGrassSampler(afGroundTCoord);
        ColorRGBA kStoneColor = rkStoneSampler(afGroundTCoord);
        ColorRGBA kBlendColor = rkBlendSampler(afBlendTCoord);

        float afOffsetCloudTCoord[2] =
        {
            afCloudTCoord[0] + afFlowDirection[0],
            afCloudTCoord[1] + afFlowDirection[1]
        };
        ColorRGBA kCloudColor = rkCloudSampler(afOffsetCloudTCoord);

        float fStoneWeight = Mathf::Pow(kBlendColor.R(), fPowerFactor);
        float fGrassWeight = 1.0f - fStoneWeight;
        ColorRGBA kPixelColor = kCloudColor*(fGrassWeight*kGrassColor +
                                             fStoneWeight*kStoneColor);

        return kPixelColor;
    }

    WM4_IMPLEMENT_PPROGRAM(BlendedTerrain);
    WM4_REGISTER_PPROGRAM(BlendedTerrain);
}

```

I mentioned that I copy the `dx9.wmsp` file and edit the copy to produce the software renderer's version of the shader program. One issue to be aware of is that Direct3D does not have a representation for a 1D texture. The line of code in the `dx9.wmsp` file related to the 1D texture is the sampler declaration

```
//var sampler2D BlendSampler : : texunit 2 : 8 : 1
```

Notice that this is a `sampler2D`, even though the Cg program listed it as a `sampler1D`. The Cg compiler converts this to a `sampler2D` when compiling for Direct3D. The software renderer expects the sampler to be of the type specified by Cg, so the corresponding shader code is

```
// var sampler1D BlendSampler    texunit 2
```

The macros used at the end of the pixel program code block are necessary for the pixel program to be registered with the software renderer. The registration guarantees that the software renderer can locate the pixel program during application execution.

Because of the nonstandard extensions used for the software shader programs, you need to tell your development environment to compile these using the C++ compiler. In Microsoft's Visual Studio, add the two files to the project. For the software renderer build configurations, select each of the files and then launch the properties dialog. One of the options in the dialog is to specify the tool used to process the file. Select it to be the C++ compiler.

A.2 CREATING THE GEOMETRIC DATA

The height field is created to have an up direction of $(0, 0, 1)$. The function creating the height field is shown next. The terrain covers the xy -rectangle $|x| \leq e_0$ and $|y| \leq e_1$, where e_0 is represented by `fxExtent` and e_1 is represented by `fYExtent`. The inputs `iXSamples` and `iYSamples` specify how many grid cells to create in the rectangle with two triangles per cell. The heights are all set to zero initially but modified later to nonzero values.

```
void BlendedTerrain::CreateHeightField ()
{
    Attributes kAttr;
    kAttr.SetPChannels(3);    // position (x,y,z)
    kAttr.SetTChannels(0,2);  // grass and stone (s,t) tcoords
    kAttr.SetTChannels(1,1);  // blending (s) tcoord
    kAttr.SetTChannels(2,2);  // cloud (s,t) coords

    const int iXSamples = 64, iYSamples = 64;
    const float fxExtent = 8.0f, fYExtent = 8.0f;
    int iVQuantity = iXSamples * iYSamples;
```



```

int iTQuantity = 2 * (iXSamples - 1) * (iYSamples - 1);
VertexBuffer* pkVB = WM4_NEW VertexBuffer(kAttr,iTQuantity);
IndexBuffer* pkIB = WM4_NEW IndexBuffer(3*iTQuantity);

// Generate the geometry for a flat height field.
float fInv0 = 1.0f/(iXSamples - 1.0f);
float fInv1 = 1.0f/(iYSamples - 1.0f);
float fU, fV;
int i, i0, i1;
for (i1 = 0, i = 0; i1 < iYSamples; i1++)
{
    fV = i1*fInv1;
    Vector3f kYTmp = ((2.0f*fV - 1.0f)*fYExtent)*Vector3f::UNIT_Y;
    for (i0 = 0; i0 < iXSamples; i0++)
    {
        fU = i0*fInv0;
        Vector3f kXTmp = ((2.0f*fU - 1.0f)*fXExtent)*Vector3f::UNIT_X;
        pkVB->Position3(i) = kXTmp + kYTmp;
        Vector2f kTCoord(fU,fV);
        pkVB->TCoord2(0,i) = kTCoord;
        pkVB->TCoord1(1,i) = 0.0f;
        pkVB->TCoord2(2,i) = kTCoord;
        i++;
    }
}

// Generate the index array for a regular grid of squares,
// each square a pair of triangles.
int* aiIndex = pkIB->GetData();
for (i1 = 0, i = 0; i1 < iYSamples - 1; i1++)
{
    for (i0 = 0; i0 < iXSamples - 1; i0++)
    {
        int iV0 = i0 + iXSamples * i1;
        int iV1 = iV0 + 1;
        int iV2 = iV1 + iXSamples;
        int iV3 = iV0 + iXSamples;
        aiIndex[i++] = iV0;
        aiIndex[i++] = iV1;
        aiIndex[i++] = iV2;
        aiIndex[i++] = iV0;
        aiIndex[i++] = iV2;
        aiIndex[i++] = iV3;
    }
}

```

```

// Set the heights based on a precomputed height field.
m_spkHeight = Image::Load("BTHightField");
unsigned char* pucData = m_spkHeight->GetData();
for (i = 0; i < iVQuantity; i++, pucData += 3)
{
    unsigned char ucValue = *pucData;
    float fHeight = ((float)ucValue)/255.0f;
    float fPerturb = 0.05f*Mathf::SymmetricRandom();
    pkVB->Position3(i).Z() = 3.0f*fHeight + fPerturb;
    pkVB->TCoord2(0,i) *= 8.0f;
    pkVB->TCoord1(1,i) = fHeight;
}

m_spkHeightField = WM4_NEW TriMesh(pkVB,pkIB);
m_spkScene->AttachChild(m_spkHeightField);
}

```

The heights for the terrain are stored in the red channel of a gray-scale RGB image. This image is loaded from disk and the heights are assigned accordingly. The height image is shown in Figure A.1.

The application function `CreateScene` creates a scene graph with two children, the first a sky dome that has already been created and loaded from disk and the second the height field constructed as shown previously. The sky dome is a hemisphere and has a texture whose image is shown in Figure A.2. The 1D height texture used for blending grass and stone is a linear ramp in gray-scale color:

```

const int iHSize = 256;
unsigned char* aucData = WM4_NEW unsigned char[3*iHSize];
int i;
for (i = 0; i < iHSize; i++)
{
    aucData[3 * i + 0] = i;
    aucData[3 * i + 1] = i;
    aucData[3 * i + 2] = i;
}
m_spkBlend = WM4_NEW Image(Image::IT_RGB888,iHSize,aucData,'BTBlend');

```

A.3 A CLASSLESS SHADER EFFECT

To create a shader effect for the terrain without writing source code for a class to manage the effect's data, use the following code:

```

VertexShader* pkVShader = WM4_NEW VertexShader("BlendedTerrain");
PixelShader* pkPShader = WM4_NEW PixelShader("BlendedTerrain");

```

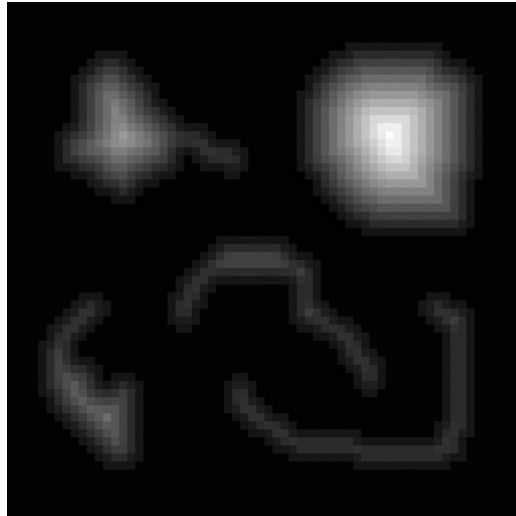


Figure A.1 The height field for the terrain is stored in a gray-scale RGB image.



Figure A.2 The image associated with the sky dome.

```

pkPShader->SetTextureQuantity(4);
pkPShader->SetImageName(0,"BTGrass");
pkPShader->SetImageName(1,"BTStone");
pkPShader->SetImageName(2,"BTBlend");
pkPShader->SetImageName(3,"BTCloud");
pkPShader->GetTexture(0)->SetFilterType(Texture::LINEAR_LINEAR);
pkPShader->GetTexture(0)->SetWrapType(0,Texture::REPEAT);
pkPShader->GetTexture(0)->SetWrapType(1,Texture::REPEAT);
pkPShader->GetTexture(1)->SetFilterType(Texture::LINEAR_LINEAR);
pkPShader->GetTexture(1)->SetWrapType(0,Texture::REPEAT);
pkPShader->GetTexture(1)->SetWrapType(1,Texture::REPEAT);
pkPShader->GetTexture(2)->SetFilterType(Texture::LINEAR);
pkPShader->GetTexture(2)->SetWrapType(0,Texture::CLAMP_EDGE);
pkPShader->GetTexture(3)->SetFilterType(Texture::LINEAR_LINEAR);
pkPShader->GetTexture(3)->SetWrapType(0,Texture::REPEAT);
pkPShader->GetTexture(3)->SetWrapType(1,Texture::REPEAT);

ShaderEffect* pkEffect = WM4_NEW ShaderEffect(1);
pkEffect->SetVShader(0,pkVShader);
pkEffect->SetPShader(0,pkPShader);

m_pkRenderer->LoadResources(pkEffect);
Program* pkProgram = pkEffect->GetVProgram(0);
pkProgram->GetUC("FlowDirection")->SetDataSource(m_afFlowDirection);
pkProgram = pkEffect->GetPProgram(0);
pkProgram->GetUC("PowerFactor")->SetDataSource(m_afPowerFactor);

m_spkHeightField->AttachEffect(pkEffect);

```

The VertexShader and PixelShader objects are created. The input string is the name used to locate the shader programs. The vertex shader is associated with `v_BlendedTerrain.wmsp` and the pixel shader is associated with `p_BlendedTerrain.wmsp`.

The pixel shader requires four textures. The texture images named by BTGrass, BTStone, and BTCloud images are found on disk by the image catalog system. These are shown in Figure A.3. The image BTBlend is already in the catalog, placed there when the Image object `m_spkBlend` was created. It does not exist on disk. The filter types and wrap types for the texture coordinates are specified for all images.

A ShaderEffect object is created. The input parameter 1 indicates that the effect uses a single pass. The vertex and pixel shaders are then attached to the shader effect.

Both the vertex and pixel programs have inputs that are user-defined constants. The vertex program has an input, FlowDirection, which is used to offset the cloud texture coordinates. The input is time-varying, modified by the application code

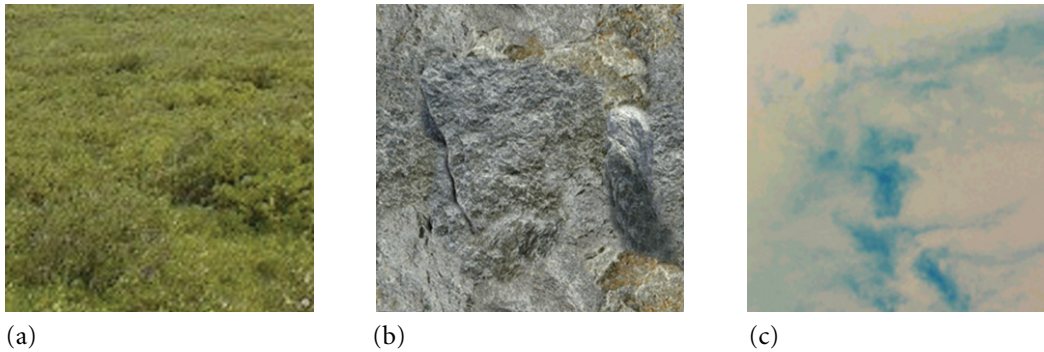


Figure A.3 (a) The grass image `BTGrass.wmi f`. (b) The stone image `BTStone.wmi f`. (c) The cloud image `BTCloud.wmi f`.

itself. The pixel program has an input, `PowerFactor`, which is used to adjust the blending coefficients for the grass and stone. The application provides storage for both user-defined constants, `m_afFlowDirection` and `m_afPowerFactor`. These are arrays of four floating-point numbers each, since the shader constants are stored in 4-float registers. Only the first two components of `m_afFlowDirection` are used and only the first component of `m_afPowerFactor` is used. Although not necessary, it is convenient to redirect the shader program to use these storage locations rather than the default ones created by the `Program` base class. The user-defined constants are looked up by name, and the function `SetDataSource` is used for the redirection. In order for this to work, it is necessary that the shader programs actually be loaded from disk into system memory (via the shader program catalogs). It is therefore required that you force a load of the effect by calling the renderer's `LoadResources` function.

A.4 CREATING A CLASS DERIVED FROM `SHADEREFFECT`

The code block for a classless construction of a shader effect can be encapsulated in a class, which in this application is named `BlendedTerrainEffect`. If you prefer, think of the class as something in your own fixed-function pipeline. It must be compiled and linked into an application to be used. The classless approach allows you to create effects at run time, which is more flexible but requires you to make all the appropriate function calls to set up the effect.

The class declaration is

```
class BlendedTerrainEffect : public ShaderEffect
{
    WM4_DECLARE_RTTI;
    WM4_DECLARE_NAME_ID;
    WM4_DECLARE_STREAM;

public:
    BlendedTerrainEffect (const char* acGrassName,
        const char* acStoneName, const char* acBlendName,
        const char* acCloudName);
    virtual ~BlendedTerrainEffect ();

    // for the vertex program
    void SetFlowDirection (const Vector2f& rkFlowDirection);
    Vector2f GetFlowDirection () const;

    // for the pixel program
    void SetPowerFactor (float fPowerFactor);
    float GetPowerFactor () const;

protected:
    // streaming
    BlendedTerrainEffect ();

    // Set the user-defined constants to use local storage.
    virtual void OnLoadPrograms (int iPass, Program* pkVProgram,
        Program* pkPPProgram);

    // The flow direction is stored in locations 0 and 1.
    // The others are unused.
    float m_afFlowDirection[4];

    // The power factor is stored in location 0. The
    // others are unused.
    float m_afPowerFactor[4];
};
```

The constructor accepts the string names of the four texture images to be used by the effect. The texture setup occurs within the constructor. The shader constants are stored as class data members. The redirection mentioned earlier occurs in the function `OnLoadPrograms`, which is called the first time the programs are required by the

application. You can also make this happen with a call to the renderer's `LoadResources` function, but if you do not, the renderer's `ApplyEffect` function will load the shader programs and call `OnLoadPrograms`. The binding here is as late as possible—just before the effect is used for drawing.

A.5 DYNAMIC UPDATES FOR THE SHADER CONSTANTS

The sample application has a function, `UpdateClouds`, which updates the `FlowDirection` vertex program constant. This function is called in the idle loop. The visual result is that the texture coordinates are translated each frame, giving the appearance of the cloud shadows moving over the terrain.

The `OnKeyDown` callback allows you to press the plus (+) key and minus (−) key to change the `PowerFactor` pixel program constant. The blending of the grass and stone textures (without the cloud shadows) is

$$C_{\text{blend}} = (1 - h^f) C_{\text{grass}} + h^f C_{\text{stone}}$$

where $h \in [0, 1]$ is the red channel of the sampled 1D texture image and where $f > 0$ is the power factor. When $f = 1$, the blending uses linear interpolation.

The redirection of the storage for the shader constants allows you to change the constants without having to worry about any details of how those constants are fed to the shader programs.

When you run the application, the flow direction is chosen so that the clouds move from right to left. The sky dome is rotated at a rate that makes it appear as though its clouds are moving and are the ones casting shadows on the ground. Screen captures are shown in Figure A.4.



Figure A.4 Two screen captures from the BlendedTerrain sample application. The power factor is $1/2$. Notice that the cloud shadows are different in the two images.

REFERENCES

- [AJ97] E. Andres and M. A. Jacob. The discrete analytical hyperspheres. *IEEE Transactions on Visualization and Computer Graphics*, 3(1):75–86, 1997.
- [AM01] Timo Aila and Ville Miettinen. *SurRender Umbra Reference Manual*, vol. 2.3, 2001.
- [And94] E. Andres. Discrete circles, rings and spheres. *Computer and Graphics*, 18(5):695–706, 1994.
- [Arv91] James Arvo, ed. *Graphics Gems II*. Academic Press, San Diego, CA, 1991.
- [AS65] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, 1965.
- [Bar01] David Baraff. Physically Based Modeling: Rigid Body Simulation. www.pixar.com/companyinfo/research/pbm2001/notesg.pdf, 2001, 68 pages.
- [BF01] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*, 7th ed., Brooks/Cole, Belmont, CA, 2001.
- [Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. *ACM Computer Graphics (Proceedings of SIGGRAPH 1977)*, July 1977, pp. 192–198.
- [Boo87] Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems*. Pearson Benjamin Cummings, San Francisco, CA, 1987.
- [Bre65] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.
- [Cam97] Stephen Cameron. Enhancing GJK: Computing minimum and penetration distances between convex polyhedra. In *Proceedings of the IEEE Int’l Conference on Robotics and Automation*, pp. 3112–3117, 1997.
- [CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
- [COM98] Jonathan D. Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplifications. In *Proceedings of SIGGRAPH*, 1998, pp. 115–122.
- [Cor] Microsoft Corporation. The DirectX Software Development Kit. www.microsoft.com.
- [CRE01] Elaine Cohen, Richard F. Riesenfeld, and Gershon Elber. *Geometric Modeling with Splines: An Introduction*. A. K. Peters, Natick, MA, 2001.
- [CVM⁺96] J. D. Cohen, A. Varshney, D. Manocha, G. Turk, et al. Simplification envelopes. In *Proceedings of SIGGRAPH*, 1996, pp. 119–128.

- [DK90] D. P. Dobkin and D. G. Kirkpatrick. Determining the separation of preprocessed polyhedra—a unified approach. *Proceedings of the 17th Int'l Colloq. Automata Lang. Program, Lecture Notes in Computer Science*, 43:400–413, 1990.
- [DSS88] H. Das, J.-J. E. Slotine, and T. B. Sheridan. Inverse kinematic algorithms for redundant systems. *IEEE International Conference on Robotics and Automation*, 1988, pp. 43–48.
- [DWS⁺97] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, et al. Roaming terrain: Real-time optimally adapting meshes. *IEEE Visualization 1997*, pp. 81–88.
- [Ebe03] David Eberly. *Game Physics*. Morgan Kaufmann, San Francisco, CA, 2003.
- [Ede87] H. Edelsbrunner. *Algorithms in Computational Geometry*. Springer-Verlag, Heidelberg, Germany, 1987.
- [EM85] H. Edelsbrunner and H. A. Maurer. Finding extreme points in three dimensions and solving the post-office problem in the plane. *Inform. Process. Lett.*, 21:39–47, 1985.
- [Eng02] Wolfgang F. Engel, ed. *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*. Wordware, Plano, TX, 2002.
- [Eng03] Wolfgang F. Engel, ed. *ShaderX2: Shader Programming Tips & Tricks*. Wordware, Plano, TX, 2003.
- [Eng04] Wolfgang F. Engel, ed. *ShaderX3: Advanced Rendering with DirectX and OpenGL*. Charles River Media, Hingham, MA, 2004.
- [Eng06] Wolfgang F. Engel, ed. *ShaderX4: Lighting and Rendering*. Charles River Media, Hingham, MA, 2006.
- [Eri04] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, San Francisco, CA, 2004.
- [Far90] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, San Diego, CA, 1990.
- [Far99] Gerald Farin. *NURBS: From Projective Geometry to Practical Use*. A. K. Peters, Natick, MA, 1999.
- [Fer04] Randima Fernando, ed. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley, Boston, MA, 2004.
- [FG03] Kaspar Fischer and Bernd Gärtner. The smallest enclosing ball of balls: Combinatorial structure and algorithms. In *Annual Symposium on Computational Geometry: Proceedings of the Nineteenth Conference on Computational Geometry*, 2003, pp. 292–301.
- [FKN79] Henry Fuchs, Zvi Kedem, and Bruce Naylor. Predetermining visibility priority in 3D scenes. In *Proceedings of SIGGRAPH*, 1979, pp. 175–181.
- [FKN80] Henry Fuchs, Zvi Kedem, and Bruce Naylor. On visible surface generation by a priori tree structures. In *Proceedings of SIGGRAPH*, 1980, pp. 124–133.

- [FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*, 2nd ed. Addison-Wesley, Reading, MA, 1990.
- [GF90] E. G. Gilbert and C-P. Foo. Computing the distance between general convex objects in three-dimensional space. *IEEE Transactions on Robotics and Automation*, 6(1):53–61, 1990.
- [GH97] Michael Garland and Paul Heckbert. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH*, 1997, pp. 209–216.
- [GH98] Michael Garland and Paul Heckbert. Simplifying surfaces with color and texture using quadric error metrics. *IEEE Visualization*, 1998, pp. 263–269.
- [GJK88] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between objects in three-dimensional space. *IEEE Journal Robotics and Automation*, vol. 4:193–203, 1988.
- [GL93] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*, 2nd ed. Johns Hopkins University Press, Baltimore, MD, 1993.
- [Gla90] Andrew S. Glassner, ed. *Graphics Gems I*. Academic Press, San Diego, CA, 1990.
- [GLM96] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings of SIGGRAPH*, 1996, pp. 171–180.
- [Hec94] Paul Heckbert, ed. *Graphics Gems IV*. Academic Press, San Diego, CA, 1994.
- [HJ85] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, Cambridge, UK, 1985.
- [Hop96a] Hugues Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH*, 1996, pp. 99–108.
- [Hop96b] Hugues Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH*, 1996, pp. 189–198.
- [Kal] Michael Kallay. Computing moment of inertia. Private correspondence, November 29, 2004.
- [KB86] Doris H. U. Kochanek and Richard H. Bartels. Interpolating splines with local tension, continuity, and bias control. *ACM SIGGRAPH Course Notes 22, Advanced Computer Animation*, 1986.
- [Kir83] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12:28–35, 1983.
- [Kir92] David Kirk, ed. *Graphics Gems III*. Academic Press, San Diego, CA, 1992.
- [Knu73] Donald E. Knuth. *The Art of Computer Programming, Volumes 1–3*. Addison-Wesley, Boston, MA, 1973.

- [Lan98] Jeff Lander. Oh my god, i inverted kine. *Game Developer Magazine*, 1998, pp. 9–14.
- [LB84] Y-D. Liang and B. A. Barsky. A new concept and method for line clipping. *ACM Transactions on Graphics*, 3(1):1–22, 1984.
- [LE97] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of ACM SIGGRAPH 1997*, pp. 199–208.
- [Lev00] Ron Levine. Collision of Moving Objects. www.sourceforge.net (on the game developer algorithms list), November 2000.
- [LH04] F. Losasso and H. Hoppe. Geometry clipmaps: Terrain rendering using nested regular grids. In *Proceedings of ACM SIGGRAPH 2004*, pp. 769–776.
- [Lib] Boost C++ Libraries. Computing Moment of Inertia. www.boost.org/.
- [Lim01] Ravenbrook Limited. The Memory Management Reference. www.memory-management.org, 2001.
- [LKR⁺96] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, et al. Real-time, continuous level of detail rendering of height fields. In *Proceedings of SIGGRAPH*, 1996, pp. 109–118.
- [LLM05] Stanley B. Lippman, J  see Lajoie, and Barbara E. Moo. *C++ Primer*, 4th ed. Addison-Wesley, Reading, MA, 2005.
- [Lom03] Chris Lomont. Fast Inverse Square Root. www.math.purdue.edu/~clomont/Math/Papers/2003/InvSqrt.pdf, 2003.
- [LRC⁺03] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, et al. *Level of Detail for 3D Graphics*. Morgan Kaufmann, San Francisco, CA, 2003.
- [LT98] Peter Lindstrom and Greg Turk. Fast and memory efficient polygonal simplification. *IEEE Visualization*, 1998, pp. 279–286.
- [MB05] Tom McReynolds and David Blythe. *Advanced Graphics Programming Using OpenGL*. Morgan Kaufmann, San Francisco, CA, 2005.
- [Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. International Series in Computer Science, C. A. R. Hoare, ed., Prentice Hall, Englewood Cliffs, NJ, 1988.
- [MH02] Tomas M  ller and Eric Haines. *Real-Time Rendering*, 2nd ed. A. K. Peters Ltd., Natick, MA, 2002.
- [Mir96] Brian Mirtich. Fast and accurate computation of polyhedral mass properties. *Journal of Graphics Tools*, 1(2):31–50, 1996.
- [M  l97] Tomas M  ller. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997.
- [MT97] Tomas M  ller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

- [NN90] Zoran R. Novaković and Bojan Nemec. A solution of the inverse kinematics problem using the sliding mode. *IEEE Transactions on Robotics and Automation*, 6(2):247–252, 1990.
- [O’R85] J. O’Rourke. Finding minimal enclosing boxes. *Int’l Journal of Comput. Inform. Sci.*, 14:183–199, June 1985.
- [O’R98] Joseph O’Rourke. *Computational Geometry in C*, 2nd ed. Cambridge University Press, Cambridge, UK, 1998.
- [Pae95] Alan Paeth, ed. *Graphics Gems V*. Academic Press, San Diego, CA, 1995.
- [Pau06] Brian Paul. The Mesa 3D Graphics Library. www.mesa3d.org, 2006.
- [PFTV88] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, UK, 1988.
- [Pha05] Matt Pharr, ed. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, Boston, MA, 2005.
- [Pho00] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):335–342, July 2000.
- [Pro06] Virtual Terrain Project. www.vterrain.org/LOD/Papers/, 2006.
- [PS85] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, Heidelberg, Germany, 1985.
- [PZB90] C. Phillips, J. Zhao, and N. Badler. Interactive real-time articulated figure manipulation using multiple kinematic constraints. *SIGGRAPH I3D Symposium*, 24(2):245–250, 1990.
- [RB93] Jared Rossignac and Paul Borrel. Multiresolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics: Methods and Applications*, B. Falcidieno and T. Kunii, eds. Springer-Verlag, Heidelberg, Germany, 1993, pp. 455–463.
- [RM03] Andrew Rollings and Dave Morris. *Game Architecture and Design: A New Edition*. Peachpit Press, Berkeley, CA, 2003.
- [Rog01] David F. Rogers. *An Introduction to NURBS with Historical Perspective*. Morgan Kaufmann, San Francisco, CA, 2001.
- [RWPD05] Erik Reinhard, Greg Ward, Sumanta Pattanaik, and Paul Debevec. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann, San Francisco, CA, 2005.
- [SE02] Philip J. Schneider and David Eberly. *Geometric Tools for Computer Graphics*. Morgan Kaufmann, San Francisco, CA, 2002.
- [Sha99] Brian Sharp. Optimizing curved surface geometry. *Game Developer Magazine*, 1999, pp. 40–48.

- [Spe06] Henry Spencer. *getopt* command-line parser. www.lysator.liu.se/c/henry/l, 2006.
- [SS87] Lorenzo Sciavicco and Bruno Siciliano. A dynamic solution to the inverse kinematic problem for redundant manipulators. In *IEEE Int'l Conference on Robotics and Automation*, 1987, pp. 1081–1086.
- [Str00] Bjarne Stroustrup. *The C++ Programming Language*, 3rd ed. Addison-Wesley, Reading, MA, 2000.
- [SWND05] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2*. Addison-Wesley, Boston, MA, 2005.
- [SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In *Proceedings of SIGGRAPH*, 1992, pp. 65–70.
- [Tou84] G. T. Toussaint. An optimal algorithm for computing the minimum vertex distance between two crossing convex polygons. *Proceedings of IEEE Int'l Conference on Pattern Recogn.*, 1984, pp. 465–467.
- [vdB99] Gino van den Bergen. A fast and robust GJK implementation for collision detection of convex objects. *Journal of Graphics Tools*, 4(2):7–25, 1999.
- [vdB03] Gino van den Bergen. *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann, San Francisco, CA, 2003.
- [Wad] Bretton Wade. BSP Tree Frequently Asked Questions. www.faqs.org/faqs/graphics/bsptree-faq/.
- [WC91] Li-Chun Wang and Chih Cheng Chen. A combined optimization method for solving the inverse kinematics problem of mechanical manipulators. *IEEE Transactions on Robotics and Applications*, 7(4):489–499, 1991.
- [Wel91] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). *Lecture Notes in Computer Science*, 555:359–370, 1991.
- [Wel93] Chris Welman. “Inverse Kinematics and Geometric Constraints for Articulated Figures.” Master’s thesis, Simon Fraser University, Burnaby, Canada, 1993.
- [WG95a] Chionh Eng Wee and Ronald N. Goldman. Elimination and resultants, Part 1: Elimination and bivariate resultants. *IEEE Computer Graphics and Applications*, 1995, pp. 69–77.
- [WG95b] Chionh Eng Wee and Ronald N. Goldman. Elimination and resultants, Part 2: Multivariate resultants. *IEEE Computer Graphics and Applications*, 1995, pp. 60–69.
- [Wil83] Lance Williams. Pyramidal parametrics. *Computer Graphics*, 7(3):1–11, 1983.

- [WW92] Alan Watt and Mark Watt. *Animation and Rendering Techniques: Theory and Practice*. ACM Press, New York, 1992.
- [ZB94] Jianmin Zhao and Norman I. Badler. Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Transactions on Graphics*, 13(4):313–336, 1994.

INDEX

NUMBERS

3D objects, 7
3D picking, 247
3-tuples, 8, 17
4-tuples, 17, 40, 129
16-byte data alignment, 17

A

Absolute dot product, 532
Abstract base class, 455–461
 defined, 457
 Find function, 459–461
 implementation, 458–459
 implementation for
 estimating pseudodistance
 derivative, 461
 interface, 455–457
 See also Classes
Abstract interfaces
 bounding volume, 248–251
 data association and, 248
Abstract rendering API,
 175–194
 buffer clearing, 178–179
 camera management,
 176–177
 construction/destruction,
 175–176
 global-state management,
 177–178
 miscellaneous operations,
 180–182
 object drawing, 179–180
 resource management,
 182–194
 text and 2D drawing, 180

Accelerated graphics port
 (AGP) memory, 174, 183
 use, 194
 VRAM transfer speed, 174,
 183
 See also Memory
Acceleration function, 509,
 512, 515, 519, 521
Active intervals, 466
Ada, 790
AddInitializer function, 826
Additive blending, 119
Adjacent indices, 519
AdjustVerticalDistance
 function, 479, 480
Affine algebra, 15, 240
Affine transformations, 29–31
 block-matrix form, 41
 characterization, 43
 conditions, 29
 defined, 29
 between points, 29–30
 translation, 30
 See also Transformations
Aliasing, reducing, 108
Alignment
 billboard, 379
 16-byte data, 17
Allocate function, 515, 877,
 879, 880
Allocation
 with buddy-system methods,
 891–894
 pseudocode, 884–886,
 892–893
 with segregated-storage
 methods, 895
 with sequential-fit methods,
 882–891

 speed, 882
 See also Deallocation;
 Memory
Alpha blending, 170
 code in ApplyPixelShader,
 170
 defined, 117
 disabled, 170
 equation, 117
 in pixel pipeline, 166
Alpha channels, 94
Alpha testing
 applications, 121–122
 defined, 120
 in pixel pipeline, 166
Ambient light, 93, 95, 262, 303
 global, 99
 plane and sphere
 illumination, 906,
 907, 908, 909
 shader program, 901
 See also Lighting; Lights
Angular momentum, 523–524
 equations of motion, 524
 rigid bodies, 523
 See also Linear momentum
Angular velocity, 524
Animation
 controller-based, 315–351
 keyframe, 317–320
 skin-and-bones, 347–349
Anisotropic filtering, 114, 115
Application class, 836–839
 defined, 836
 interface, 836–837
Application layer, 831–872
 Application class, 836–839
 command line parameters,
 832–836

- Application layer (*continued*)
 - console, 831
 - ConsoleApplication class, 839–842
 - engine management, 867–872
 - event handlers, 867
 - features, 831
 - idle loop, 867
 - initialization/termination, 831
 - Wild Magic, 831
 - window, 831
 - WindowApplication class, 842–849
 - WindowApplication3 class, 849–866
 - See also* Object-oriented infrastructure
 - Application library, 831
 - Application objects, 839
 - Application programming
 - interfaces (APIs), 125
 - abstract rendering, 175–194
 - conventions, 128–145
 - fast computations, 143–145
 - graphics, 125
 - matrix composition, 134
 - matrix representation and storage, 129–133
 - projection matrices, 136–139
 - rotations, 140–143
 - view matrices, 134–136
 - window handedness, 139–140
 - Applications
 - alpha testing, 121–122
 - console, 839–842
 - picking, 478–479
 - types, 832
 - windowed, 842–849
 - See also specific applications*
 - ApplyEffect function, 179, 180, 199–200, 201, 970
 - ApplyForward function, 240
 - ApplyInverse function, 240
 - ApplyPixelShader function, 92, 166, 167, 170
 - ApplyStencilCompare function, 167–168
 - Arc length
 - parameterized by, 542
 - reparameterization by, 541, 543–544
 - specifying, 543
 - subdivision by, 566–567
 - Artificial intelligence (AI)
 - engine, 870
 - Assignment operator, 805
 - AttachChild function, 230
 - AttachLight function, 265
 - AttachOutside function, 370
 - Attenuation, spot, 97–98
 - Attributes
 - comparison, 215
 - shader, 206
 - surface, 92
 - vertex, 92–125
 - Attributes class, 206
 - Axis-aligned bounding boxes (AABBs)
 - collision culling with, 465–472
 - computation queries, 465
 - defined, 534
 - intersecting boxes, 472
 - intersecting intervals, 466–471
 - intersecting rectangles, 471
 - nonintersecting, 465
 - updated, 466
 - Axis-aligned boxes, 472
 - Axis-aligned ellipses
 - code, 86–87
 - decision variable, 85–86
 - implicit definition, 85
 - points to, 672–673
 - rasterization, 85–87
 - See also* Ellipses
 - Axis-aligned rectangles, 471
- ## B
- Back-face culling, 67–70, 151–154
 - coordinate systems and, 67–68
 - default, 152
 - defined, 67
 - See also* Culling
 - Back-facing triangles, 69
 - detection, 158
 - rasterizer test, 159
 - See also* Triangles
 - Backward differences, 453–454
 - defined, 453
 - next iterate based on, 454
 - Banded matrix, 324
 - Barycentric coordinates, 534, 613
 - computation, 652
 - for intersection point, 651
 - BasisFunction class, 557, 582
 - BeginScene function, 179, 869
 - Best fit policy, 890
 - Bézier curves, 545–548
 - Bernstein form, 545
 - defined, 545
 - definitions, 545
 - degree elevation, 546
 - degree reduction, 546–548
 - derivative, 545
 - evaluation, 545–546
 - See also* Curves
 - Bézier rectangle patches, 574–578
 - defined, 574–575
 - degree elevation, 575–576
 - degree reduction, 576–577
 - evaluation, 575
 - Bézier triangle patches, 578–582
 - defined, 578

- degree elevation, 580
- degree reduction, 580–582
- evaluation, 578–580
- Bézout determinant, 721
- Bilinear filtering, 106, 110
- Bilinear interpolation, 104, 110
- Billboards, 378–379
 - axial alignment, 379
 - defined, 378
 - screen alignment, 379
- Binary buddy systems, 891
- Binary space partitioning
 - defined, 354
 - nodes, 358, 364
- Bisection
 - defined, 737
 - in many dimensions, 741
 - in one dimension, 737
- Bitangents, 917
- Bits, shifting, 107
- BlendedTerrainEffect class, 968–970
- Blending
 - additive, 119
 - alpha, 117, 170
 - for dark maps, 119
 - destination, coefficients, 119
 - destination, functions, 118
 - modes, 118–119
 - soft addition, 119
 - source, coefficients, 118
 - source, functions, 118
- Blinn lighting, 97
- Blockers, 376
- Block function, 878, 879
- Block-matrix form, 40–41
- Blueprints, 312, 488–489
 - abstract graph from, 313
 - defined, 489
 - room, 484
- Boost library, 801
- Border colors, 103
- Bottom plane, 47
- Boundary class, 503, 504
- Boundary polygons, 483
- Bounded memory, 896
- Bounding boxes
 - axis-aligned, 465–472
 - oriented, 436–443
 - See also* Oriented bounding boxes (OBBs)
- Bounding spheres, 278
- BoundingBox class, 248, 249
- Bounding volumes, 244–251
 - abstract interface, 248–251
 - collision determination, 247
 - culling, 245–247
 - default type, 247
 - defined, 244
 - merging, 251
 - model, 277
 - pairs, 251
 - in Spatial class, 250
 - 3D picking, 247
 - vertex data, 250
 - world, propagation, 274
 - world, updating, 277
- Boxes, 534–535, 617–627
 - axis-aligned, 472, 534
 - capsule pseudodistance, 464
 - containing points, 618–625
 - fitting based on mesh edges, 621–622
 - fitting based on mesh faces, 622–623
 - fitting based on mesh solid, 624
 - fitting points with mean and covariance, 618–621
 - intersecting, 472
 - merging, 625–627
 - minimum-volume, 624–625
 - point in, 617–618
 - sphere pseudodistance, 463
 - See also* Oriented bounding boxes (OBBs); Oriented boxes
- Brent's method, 732–733
 - bracketing, 732
 - defined, 732
 - variation, 733
- Bresenham's algorithm, 78, 82, 148
 - defined, 78
 - drawing circles and, 82
 - pixel sets, 159
- Broad phase, collision detection, 465
- B-spline basis functions, 322
 - defined, 552
 - evaluation, 333
 - recursive dependencies, 326, 554
- BSplineCurve function, 557–558
- B-spline curves, 551–560
 - closed, 558–560
 - control points, 325
 - cubic, 333
 - defined, 321
 - degree 2, 338
 - degree 3, 333–338
 - encapsulation, 557
 - evaluation, 325–333, 553–558
 - evaluation speedup, 338
 - fitting points with, 321–324
 - illustrated, 559
 - knot vectors, 552–553
 - local control, 558
 - nonuniform, 559
 - open, 559
 - periodic, 559
 - uniform, 559
 - See also* Curves
- BSplineFitBasis class, 333
- BSplineFit class, 333
- B-spline rectangle patches, 582–583
- BSplineSurface class, 582

- BspNode class, 358–359
 - constructors, 359
 - GetVisibleSet function
 - implementation, 360–363
 - BSP trees, 354–364
 - arc testing at nodes, 402
 - of children of root, 399
 - for computational solid
 - geometry operations, 357
 - constructed by recursive
 - subdivision, 397
 - construction, 355–357
 - convex region manager, 370
 - defined, 354
 - leaf nodes, 364
 - as partitioning of space, 355
 - root, 367, 398
 - sorting, 357
 - in two dimensions, 354, 355
 - usage, 357–364
 - Buddies
 - defined, 891
 - identifying, 892
 - Buddy system
 - binary, 891
 - defined, 891–894
 - methods, 891–894
 - Buffers
 - back, 179
 - clearing, 178–179, 868
 - color, 172
 - depth, 169–170, 178–179
 - displaying, 868
 - edge, 159–161
 - encapsulating, 172
 - frame, 172–173
 - front, 179
 - index, 222, 253
 - stencil, 167–168, 179
 - swapping, 179
 - vertex, 222, 282, 284
 - BuildCompatibleArray
 - function, 214
 - Bump maps, 914–923
 - defined, 914
 - normal map, 914–916, 917
 - shader programs, 919–923
 - square, 921
 - torus, 922
- C**
- Callbacks
 - defined, 845
 - execution, 846
 - mouse, 864–865
 - OnIdle, 849, 865, 867–868
 - OnInitialize, 848
 - OnKeyDown, 871, 970
 - OnKeyUp, 871
 - OnMotion, 859, 865
 - OnMouseClicked, 859, 865
 - OnPassiveMotion, 848
 - OnPrecreate, 846, 848
 - OnSpecialKeyDown, 847, 871
 - Cameras, 43–66
 - eye point, 851
 - incremental rotation, 855
 - management, 176–177
 - motion, 851–857
 - orientation, 851
 - origin, 43
 - orthographic models, 125
 - perspective model, 43–48
 - right vector, 44
 - translation speeds, 857
 - up vector, 44
 - view direction, 43
 - Camera space. *See* View space
 - Capped cones, 538
 - Capsules, 539, 627–630
 - box pseudodistance, 464
 - capsule pseudodistance, 462
 - containing points, 628–629
 - defined, 539
 - least-squares fit, 628
 - line intersection, 703–707
 - lozenge pseudodistance, 462–463
 - merging, 629–630
 - minimum of minimum-area
 - projected circles, 628–629
 - object culling by planes, 712–713
 - point in, 627–628
 - sphere pseudodistance, 462
 - triangle pseudodistance, 464
 - Carmichael and Mason bound, 739
 - Cartesian coordinates, 8
 - Cartesian space, 8, 11
 - Catalogs, 193–194
 - classes, 203
 - defined, 193
 - hash map, 203
 - inserting/removing items, 193–194
 - multiple, 194
 - Catmull-Rom spline, 562
 - Cauchy's bound, 739
 - Ceilings, 483
 - Center-direction-radius, 532
 - Centered differences, 453
 - Cg compiler, 148, 302, 963
 - Cg pixel programs, 300, 958–959
 - Cg shader programs, 164, 212, 958–959
 - Cg vertex program, 300, 958
 - Chapter summaries, this book, 3–5
 - Characteristic polynomial, 722
 - Child nodes, 229–230, 270
 - Cholesky decomposition, 324
 - Circles
 - circles to, 676–679
 - decision variable, 82, 83
 - fitting to 2D points, 727–729
 - implementation, 83
 - pixel drawing decision, 82
 - points to, 675–676

- rasterization, 82–83
- Clamped coordinates, 100–101, 102
- Clamped splines, 549, 550
- Clamp-to-edge mode, 103–104
- Classes, 790
 - abstract base, 455–461
 - container, 801
 - derived, 799, 818
- ClearBackBuffer function, 178
- ClearBuffers function, 868
- ClearStencilBuffer function, 179
- ClearZBuffer function, 178–179
- Clip coordinates, 55
- Clip matrix, 55
- Clipping, 17, 154–158
 - in clip space, 181
 - defined, 43, 66, 70
 - entry point, 154
 - plane-at-a-time, 71–74, 154
 - planes, 155
 - planes, OpenGL, 183
 - polygon-of-intersection, 74–77, 154
 - triangles, 55
 - triangles, by one edge of rectangle, 157
 - to view frustum, 70–77
- ClipPolygon function, 154–155
- Clip space, 8, 52–55
 - clipping in, 181
 - defined, 55
 - to window space transformation, 164
- ClipToWindow function, 92
- ClodMesh class, 384–385
- Closed curves, 558–560
- Closed polyline, 252
- Closed splines, 549, 550–551
- Cloud shadows, 957–958
- Coefficient matrix, 642
- Cofactors, 720
- Coherency
 - render-state, 219
 - spatial, 218, 219
 - support, 219
- Colliders class, 455–457, 461
- Collision avoidance, 392
 - application support, 481
 - defined, 481
 - pathfinding, 481–506
- Collision culling, 436
 - with axis-aligned bounding boxes, 465–472
 - defined, 436
 - See also* Culling
- Collision detection, 389–506
 - accuracy, 449
 - algorithms, 389
 - broad phase, 465
 - dynamic, 455–472
 - moving objects, 390, 444–455
 - narrow phase, 465
 - object picking, 472–480
 - predictive, 449
 - pseudodistance, 444–445
 - robust, implementation, 392
 - speed, 449
 - stationary objects, 390
- Collision determination, 247
- Color buffers, 172
- Color masking, 166, 171
- Colors
 - border, 103
 - destination, 117, 170
 - emissive, 94
 - light, 264
 - maximum number, renderers, 182
 - source, 117, 170
 - specular, 94
 - texture, 103
 - vertex, 92, 897–899
 - Wild Magic, 183
- Column-major order, 131
- Command class, 832–833
- Command-line parameters, processing, 832–836
- Commutativity, of uniform scale and rotation, 237
- Companion matrix, 738
- ComparesFavorably() function, 121
- Compatibility, 784
- Compilers
 - Cg, 148, 302, 963
 - scene graph, 305–313
- ComputeContactInformation function, 457
- ComputeEdgeBuffers function, 91, 125, 160
- ComputeInterval function, 412, 417, 419, 429
- ComputeVisibleSet function, 868
- Cones, 537–538
 - acute angle, 538
 - axis intersection, 716
 - capped, 538
 - double-sided, 538, 710
 - line intersection, 710
 - object culling by planes, 716–717
 - single-sided, 537–538, 710
 - vertex, 716
- Configure function, 302
- Conics, project to conics, 39
- Conjugate gradient search, 734–735
- Connector class, 503, 504
- ConsoleApplication class, 839–842
 - deriving class, 840
 - interface, 839–840
- Console applications, 839–842
 - function pointer, 840
 - results display, 839
 - setup, 839

- Constructors
 - BspNode class, 359
 - ConvexRegion class, 372
 - Culler class, 291
 - default, 204
 - Portal class, 372, 375
 - RigidBody class, 526
 - WindowApplication class, 844
- Contact
 - first point, 446
 - first time, 446, 448
 - first time, computing, 448–452
 - last point, 446
 - last time, 446
- Contact set
 - calculation support, 419
 - convex polygons, 418–426
 - convex polyhedra, 428–436
- Container classes, 801
- Containment methods,
 - 609–638
 - boxes, 617–627
 - capsules, 627–630
 - cylinders, 634–636
 - ellipsoids, 636–638
 - lozenges, 631–633
 - spheres, 609–617
- Continuous level of detail, 378, 380–387
 - algorithms, 386
 - recording of vertices/indices, 385–386
 - simplification with quadratic error metrics, 380–385
 - terrain, 386–387
 - See also* Level of detail (LOD)
- Contour edges, 503
- Controller-based animation, 315–351
- Controller class, 225, 316, 317
- Controllers, 225
 - index, 225
 - keyframe, 350
 - keyframe data, 309
 - morph, 350
 - pointers, 815
 - update function, 350
 - updates, 277
 - vertex, 225
 - Wild Magic support, 316
 - in world transformation
 - computation, 276
- ConvertAxesToQuaternion
 - function, 627
- Convex functions, 449
- Convexity, 453
- Convex objects
 - defined, 391
 - illustrated, 392
 - linear component
 - intersection, 681–684
 - nonintersecting, 393
- ConvexPolygon class, 405
- Convex polygons
 - with bisectors, 401
 - clipped, vertices, 156
 - clipped by edge of rectangle, 155, 156
 - contact set, 418–426
 - convex polyhedra
 - pseudodistance, 465
 - counterclockwise-ordered
 - vertices, 395
 - edge-edge contract, 406
 - edge-edge intersection, 417
 - external query, 394, 395–396
 - extrema, 394–404
 - illustrated, 396
 - intersecting, 406
 - line-segment intersection
 - calculation, 436
 - moving, testing for
 - intersection, 413–416
 - nonintersecting, 406
 - object culling by planes, 717
 - partitioning of sphere into, 398
 - projection interval, 394
 - separation of, 409, 413–418
 - spherical, 398, 401
 - stationary objects, 404–409
 - triangle fan, 158
 - vertex-edge contact, 406
 - vertex indices, 158
 - vertex-vertex contact, 406
 - See also* Polygons
- Convex polyhedra, 245
 - contact set, 428–436
 - convex polygons
 - pseudodistance, 465
 - external query, 396–401
 - extrema, 394–404
 - object culling by planes, 717
 - for point-in-spherical-polygon test, 400
 - projection interval, 394
 - separation of, 409–410, 427–428
 - stationary objects, 409–412
 - See also* Polyhedra
- ConvexPolyhedron class, 409, 429
- Convex region
 - drawing routine, 373
 - manager, 370, 373
 - portal update initiation, 372
- ConvexRegion class, 371–372
 - constructor, 372
 - interface, 371–372
- ConvexRegionManager class, 369–372
- Convex sets, 450
- Coordinate frames, 542
- Coordinates
 - barycentric, 534, 613
 - clamped, 100–101, 102
 - clip, 55
 - integer-valued, 153
 - inter-valued, 153
 - normalized, 56, 473
 - repeated, 101, 102

- texture, 100–104
- view, 51
- window, 57, 153, 163
- world, 482, 523, 860
- Coordinate systems, 7, 9–10
 - back-face culling and, 67–68
 - Direct3D documentation, 10–11
 - elements, 9–10
 - left-handed, 10, 12, 13, 929
 - right-handed, 10, 12, 13
 - standard, 8, 9
 - vector representation, 13
- CopyFrom function, 250, 251
- Correctness, 784
- Cosine, 756
- Covariance matrix
 - of edge masses, 622
 - scaling, 623
- CPUs, 126
 - AMD, 126, 128
 - Intel, 128
- Cracking, 82
 - complicated, 598
 - no, 598
 - rectangle patch subdivision, 597–602
 - subdivision containing, 597
- Crank-Nicholson method, 752
- Create function, 248, 249
- CreateNormalMap tool, 917
- CreateScene function, 965
- Cross products, 10–14
 - calculation summary, 14
 - Direct3D computation, 14
- Cube maps, 929–932
 - creating, 932
 - defined, 929
 - lookup, 929
 - normalization, 923
 - sampler, 929
 - shader programs, 929–932
- Culler class, 289, 291–293, 363
 - constructor, destructor, 291
- culling planes in world
 - coordinates, 291
 - defined, 291
 - interface, 292–293
 - modification, 376
 - Spatial class interaction, 293
- Culling
 - back-face, 67–70, 151–154
 - back-facing triangles, 55
 - bounding volumes, 245–247
 - collision, 436, 465–472
 - defined, 43, 66
 - exact test, 66
 - hierarchical, 293–296
 - inexact pseudocode, 245–246
 - inexact test, 66–67
 - object, 66–67
 - objects by planes, 710–717
 - occlusion, 368, 375–376
 - output, 297
 - plane-at-a-time, 245, 246
 - purpose, 245
 - sorted, 296–297
 - in Wild Magic, 289
- Curve masses, 510–513
 - defined, 510
 - deformable, 511
 - illustrated, 511
 - See also* Mass-spring systems
- Curves, 541–571
 - bending, 541
 - Bézier, 545–548
 - B-spline, 551–560
 - clamped splines, 549, 550
 - classes, 542
 - closed, 558–560
 - closed splines, 549, 550–551
 - cubic polynomial, 568
 - curvature, 541, 542
 - natural splines, 549, 550
 - normal, 543
 - NURBS, 542, 560–562
 - object orientation, 570–571
 - parametric, 542
- parametric subdivision, 566–570
- planar, 542
- reparameterization by arc
 - length, 541, 543–544
- space, 543
- surfaces built from, 548–587
- tension-continuity-bias
 - splines, 562–566
- torsion, 543
- Cyclic coordinate descent
 - defined, 342
 - list manipulator with
 - multiple end effectors, 346–347
 - list manipulator with one
 - end effector, 342–345
 - numerical solution by, 342–347
 - rotate to line, 343
 - rotate to plane, 343–344
 - rotate to point, 343
 - slide to line, 245
 - slide to plane, 345
 - slide to point, 344–345
 - tree manipulator, 346
- Cyclic redundancy check (CRC), 881–882
- Cylinders, 634–636
 - containing points, 634
 - finite, 537
 - height, 634
 - infinite, 537
 - input, projecting, 635
 - least-squares line contains
 - axis, 634
 - least-squares line move to
 - minimum-area center, 635
 - line intersection, 710
 - merging, 635–636
 - object culling by planes, 715
 - points in, 634
 - radius, 634, 635

Cylinder surfaces, 584–585
 defined, 584
 generalized, 585
 parameterized, 585
See also Surfaces

D

Dark maps, blending, 119

Data
 derivable members, 815
 functions and, 788–789
 geometric, 227, 963–965
 parameterized, types,
 800–801
 streaming of, 809
 Deallocate function, 515, 880
 Deallocation
 memory block, 881
 paired with allocation, 879
 pseudocode, 886–889,
 893–894
 semantics, 880
 speed, 882
See also Allocation; Memory

De Casteljau algorithm, 546,
 575

Deformable bodies, 521–522

Degree elevation
 Bézier curves, 546
 Bézier rectangle patches,
 575–576
 Bézier triangle patches, 580
 Degree reduction
 Bézier curves, 546–548
 Bézier rectangle patches,
 576–577

Depth buffers, 169–170
 clearing, 178–179
 current value, 169–170
 defined, 169
 as read-only, 170
See also Buffers

Depth range, 47
 default, modifying, 181
 defined, 47
 Depth test, 166
 Depth textures, 946
 defined, 99
 as projected texture, 946
 rendering, 946
 sampling, 172
 for shadow maps, 172
 Derivative matrix, 113, 741
 Derivative sequences, 738–739
 Derived classes, 799, 818
 Destination blending functions,
 118
 Destination colors, 117, 170
 DetachChild function, 230
 DetachOutside function, 370
 Determinants
 Bézout, 721
 coefficient matrix, 642
 round-off errors and, 643
 testing, 643
 Development platforms, 310
 Diagonal scaling matrix, 861
 Differential equations, 508,
 747–754
 elliptic, 753
 Euler’s method, 748
 hyperbolic, 752
 midpoint method, 748
 ordinary, 747–750
 parabolic, 751–752
 partial, 750–754
 Runge-Kutta fourth-order
 method, 748–749
 Runge-Kutta with adaptive
 step, 749–750
 use, 747
See also Numerical methods
 Diffuse coefficient, 96
 Diffuse lighting, 95–96
 Dijkstra’s algorithm, 492–494
 defined, 493

 for graphic vertices, 493
 Directed acyclic graphs (DAGs),
 230, 232
 Direct3D, 7, 125
 color data packaging, 214
 conventions, 128–145
 coordinate systems, 10–11
 cross-product computation,
 14
 fast computations, 144–145
 fine-grained control, 126
 left-handed window
 coordinates, 141
 matrix composition, 134
 OnLoadVProgram
 implementation, 186–187
 portability, 125–126
 projection matrix, 136
 renderer, 65, 148
 rotations, 142, 143
 vector/matrix conventions, 8
 vector-on-left, row-major
 order, 133
 view matrices, 135
 window handedness,
 139–140
 Directional lights, 303
 defined, 93
 nonlocal observer, 97
 plane and sphere
 illumination, 907,
 909
 shader program, 902–903
See also Lighting; Lights
 Directions
 Cartesian space, 8
 linearly independent, 10
 rotation, 21
 separating, 394
 shearing, 28
 DirectSound, 870
 DisableUserClipPlane
 function, 181

- Discrete level of detail, 379–380
 - center, 380
 - defined, 378
 - popping effect, 380
 - See also* Level of detail (LOD)
- Discriminant, quadratic
 - equation, 698
- Disk cache, 882
- DisplayBackBuffer function, 178, 872
- DisplayBuffer function, 868
- Distance
 - far plane, 45
 - multiple queries, 659
 - near plane, 45
 - plane to origin, 532
 - squared, 639–679
 - See also* Squared distance
- Distance methods, 390, 639–679
 - circle to circle in 3D, 676–679
 - linear component to linear component, 642–646
 - linear component to oriented box, 663–667
 - linear component to rectangle, 657–661
 - linear component to triangle, 651–657
 - line to line, 642–643
 - line to oriented box, 664–666
 - line to ray, 643–644
 - line to rectangle, 657–659
 - line to segment, 644
 - line to triangle, 651–654
 - oriented box to oriented box, 670–672
 - point to circle in 3D, 675–676
 - point to ellipse, 672–673
 - point to ellipsoid, 673–674
 - point to line, 640
 - point to linear component, 639–641
 - point to oriented box, 663
 - point to quadratic curve, 674–675
 - point to quadric surface, 674–675
 - point to ray, 640–641
 - point to rectangle, 655–657
 - point to segment, 641
 - point to triangle, 646–651
 - ray to oriented box, 666
 - ray to ray, 645
 - ray to rectangle, 659–660
 - ray to segment, 645
 - ray to triangle, 654
 - rectangle to oriented box, 669–670
 - rectangle to rectangle, 661–663
 - rectangle to triangle, 661–663
 - segment to oriented box, 666–667
 - segment to rectangle, 660–661
 - segment to segment, 645–646
 - segment to triangle, 654–655
 - triangle to oriented box, 667–668
 - triangle to rectangle, 661–663
 - triangle to triangle, 661–663
- Dithering, 166
- Division
 - deferred, 649
 - floating-point, 649
 - knots, 331
 - See also* Subdivision
- Dobkin-Kirkpatrick hierarchy, 394, 395
- Do forever loop, 452
- Doorway class, 503, 504
- Doorways
 - defined, 482
 - room multigraphs, 482–483
 - waypoints, 486
- See also* Rooms
- DoPhysical function, 869
- DoPick function, 476
 - implementation, 477–478
 - in TriMesh class, 476–477
- Double-sided cones, 538
 - defined, 538
 - line intersection, 710
 - See also* Cones
- DoVisual function, 869
- Downsampling, 108, 109
- DrawElements function, 179
- DrawFrameRate function, 868
- Draw function, 168, 179, 180, 268
 - stencil buffer and, 168
 - support, 194–195
- Drawing
 - geometric primitives, 198–199
 - lines, 78, 82, 148
 - multipass, 300, 302–304
 - multiple-effect, 304–305
 - with multitextures, 300
 - objects, 179–180
 - pass, 297–305
 - pass index, 212
 - scenes, 195–197
 - single-effect, multipass, 302–304
 - single-pass, 298–302
 - text and, 180
- DrawTriMesh function, 70, 151
 - back-facing triangle detection, 158
 - triangle fan of convex polygon, 158
 - wireframe mode and, 157–158
- Dx9Renderer class, 180
- Dynamic binding, 790
- Dynamic collision detection, 455–472
- Dynamic initialization, 825

- Dynamic lighting, 92
- Dynamic multitextures, 116–117
- Dynamic objects algorithm, 392
- Dynamic scene graphs, 307
- Dynamic typecasting
 - defined, 793
 - template functions, 796
- Dynamic updates, 317
 - for shader constants, 970
 - SurfaceMesh class support, 521
 - See also* Updates
- Dynamic visibility graph, 491–492

E

- Edge buffers, 159–161
 - algorithm, 159
 - maximum, 161
 - triangle with, 160
 - vertex colors, 161
 - See also* Buffers
- Edge intersections
 - edge-to-edge tests, 497
 - outer edges, locating, 498–501
 - rational arithmetic, 495–497
- Edge rasterizers, 159
- Edges
 - contour, 503
 - mesh, 621–622
 - outer, locating, 498–501
 - silhouette, 503
 - total mass, 622
- Edge-to-edge contact, 423
- Effect class, 266–268
 - defined, 266
 - interface for Spatial, 267–268
- Effects, 266–268
 - applying, 199–201
- fog, 166
 - global, 196, 290
 - local, 196, 222, 266
 - multipass, 188, 195
 - rendering, 304
 - single-pass, 298
 - texture, 304–305
- Efficiency, software, 784
- Eigendecomposition, 621, 781
- Eigensystems, 722–724
 - defined, 722
 - extrema of constrained quadratic forms, 723–724
 - extrema of quadratic forms, 722–723
- Eigenvectors, 722, 724
- Ellipses
 - axis-aligned, 85–87, 672
 - general, 87–89
 - integer coefficients, 84
 - mapped to ellipses, 39
 - pixel computations, 84
 - points to, 672–673
 - rasterization, 84–89
 - specifying, 84
- Ellipsoids, 535–536, 636–638
 - containing points, 637
 - defined, 535
 - fitting points with Gaussian distribution, 637
 - line intersection, 709
 - merging, 638
 - minimum-volume, 637
 - object culling by planes, 713–715
 - points, 636–637
 - points to, 673–674
 - representation, 535
- EnableUserClipPlane function, 181
- End effectors
 - defined, 339
 - list manipulator with, 342–346
- EndScene function, 869
- Engines
 - artificial intelligence (AI), 870
 - asynchronous behavior, 871–872
 - commercial, 128
 - graphics, 868–869
 - managing, 867–872
 - networking, 870
 - physics, 316, 507–528, 869–870
 - sound, 870
 - Wild Magic, 128
- Envelopes
 - construction, 494–503
 - construction run times, 503
 - defined, 494
 - extraction, 502
 - illustrated, 502
 - performance testing, 501–503
 - projection graph, 494–495
- Environment class, 503, 504
- Environments, 482
- Euler angles, 774–777
- Euler’s method, 508, 748
- Event handling, 845–849, 867
- Events
 - key press, 847
 - mouse, 848
 - triggering, 870
 - See also* Callbacks
- Evolution, 1, 2–3
- Exact culling test, 66
- Exact interpolation, 548
- ExactSphere1 function, 612, 614
- ExactSphere2 function, 613, 614
- ExactSphere3 function, 614
- ExactSphere4 function, 614
- ExcessArguments method, 834
- Exponential fog, 123

Extendability, 784
 ExternalAcceleration
 function, 513, 515,
 519, 521
 Eye point
 in view coordinates, 69
 view frustum and, 44
 Eye space. *See* View space

F

Factorization, 775
 physical simulations and, 870
 pseudocode, 776–777
 Factory function, 817
 Far plane
 defined, 43
 distance, 45
 inner-pointing, 45
 Fast function evaluation,
 754–757
 cosine, 756
 inverse square root, 754–755
 inverse tangent, 756–757
 sine, 755–756
 square root, 754–755
 tangent, 756
 See also Numerical methods
 FastNoIntersection function,
 457
 Find function, 452
 FindIntersection pseudocode,
 423, 424–426, 431–432
 Find-intersection queries, 390,
 681–684, 693–698
 line-box, 693–698
 line-capsule, 704
 line-cylinder, 710
 line-ellipsoid, 709
 line-lozenge, 708–709
 line-sphere, 699–700
 See also Queries
 FindPolygonIntersection
 routine, 426
 Finite cylinders, 537
 Finite-difference approxima-
 tion, 916
 First derivative, estimating,
 453–455
 Fit first policy, 889
 Fixed-function pipeline, 2, 173
 Fixed-point iteration, 729, 731
 Fixed up-vectors, 571
 Flat shading, 94
 Floating-point arithmetic, 238,
 452
 division, 649
 round-off errors, 153, 495
 values, 238, 452
 Floating-point window
 coordinates, 163
 Floors, 483
 Fog
 density, 122
 effects, 166
 exponential, 123
 factor, 122
 linear, 123
 range-based, 123
 volumetric, 947–950
 See also Vertex attributes
 Fog generators, 947–948
 FooterBlock class, 883, 892
 Forward differences
 defined, 453
 next iterate based on, 454,
 455
 4-tuples, 17, 40, 129
 Fractional linear transforma-
 tion, 112
 Frame buffers, 172–173
 Frame rate
 measurement, 865
 measurement, resetting, 866
 Frenet frames
 defined, 543
 orientation with, 571
 Frenet-Serret equations, 543

Fresnel factor, 935
 Front-facing triangles, 69,
 70
 Function function, 527
 Functions
 class-static, 812
 data and, 788–789
 initialization, 824
 linking, 818
 registration, 830
 See also specific functions

G

Game consoles
 memory budgets, 873–875
 processing power, 874
 Games, evolution, 1
 Garbage collection, 895–896
 implementation, 896
 incremental, 896
 Garland-Heckbert algorithm,
 380, 381
 Gaussian elimination, 720
 Gaussian quadrature, 544,
 746–747
 defined, 746
 implementation, 747
 See also Integration
 General ellipses
 algorithm cases, 87–88
 code, 89
 decision variable, 88
 rasterization, 87–89
 subarc computation, 88
 See also Ellipses
 Generalized cylinder surfaces,
 585
 Geometric data, 227
 creating, 963–965
 representation, 227
 Geometric level of detail
 (LOD), 377

- Geometric pipeline, 58
 - model triangle to be sent through, 59
 - transformation results applied during, 61
- Geometric primitives
 - collection, 195
 - drawing, 198–199
 - illustrated, 196
 - lights and, 199
 - mismatches, 215
 - topology, 199
 - Wild Magic, 195
- Geometric state, 233–259
 - bounding volumes, 244–251
 - index buffers, 233
 - transformations, 234–244
 - vertex buffers, 233
- Geometric-state updates, 268–280
 - defined, 268
 - illustrated, 273
 - See also* Updates
- Geometric types, 251–259
 - defined, 251
 - line segments, 254–256
 - particles, 258–259
 - points, 252–254
 - storage, 252
 - triangle meshes, 256–258
- Geometry, 221–222
- Geometry class, 221, 222, 226
 - geometric types, 251
 - geometric updates, 271–272
 - global state storage, 261
 - light support, 266
 - render-state updates, 280–281
 - smart pointers, 289
- Geometry clipmaps, 387
- GetAllObjectsByName function, 820, 821
- GetAttenuation function, 904
- GetChild function, 230
- GetColorMask function, 181
- GetCommentCharacter function, 182
- GetContainingRegion function, 370
- GetDirectionalLightFactors function, 903
- GetDiskUsed function, 810, 816
- GetExtension function, 182, 203
- GetExtremeIndex function, 419
- GetGlobalState function, 178
- GetIdentifier function, 185, 187
- GetIndices function, 330
- GetIntersection function, 433–435
- GetKey function, 557
- GetKnot function, 330
- GetLastError function, 834
- GetLight function, 178, 265
- GetLinkId function, 819
- GetMaxPShaderImages function, 183
- GetMaxStencilIndices function, 183
- GetMaxTCords function, 183
- GetMaxUserClipPlanes function, 183
- GetMinIndex function, 328
- GetName function, 820
- GetObjectById function, 821
- GetObjectByName function, 820, 821
- GetObjectCount function, 810
- Getopt routines, 832
- GetParent function, 229
- GetPickRay function, 474
- GetPointLightFactors function, 904
- GetPolygon function, 436
- GetProjector function, 182
- GetSegment function, 436
- GetStateType function, 261
- GetTime function, 866
- GetVertex function, 405
- GetVisibleSet function, 296, 360–363, 370, 375
- GetWorldTriangle function, 942
- GJK algorithm, 672
- Global effects, 196, 290
 - defined, 196
 - derivation, 199
 - handling, 197
 - See also* Local effects
- GlobalState class, 259–261
- Global states, 259–261
 - active, 198
 - defined, 259
 - effect on rendering, 261
 - management, 177–178
 - Set/Get functions, 178
 - stack, 283
 - storage, 260–261
 - updates, 262
 - See also* Render states
- Gloss maps, 923–926
 - defined, 923
 - shader programs, 923–926
- GNU Image Manipulation Program (GIMP), 955
- Gouraud shading, 94
- Gram-Schmidt orthonormalization, 779–780
 - defined, 779
 - QR decomposition equivalent, 780
- Graphics drivers, 7
- Graphics engine management, 868–869
- Graphics processing units (GPUs), 1, 173
- Graphics renderer, 870
- Graphics system, 2, 7–145
- Gray-scale mapping, 915
- GrowToContain function, 250, 251
- Guard bytes, 881

H

H-adjacent triangles, 604, 605, 606
 Half-angle rotation, 766
 Hardware rendering, 173–175
 Hash maps, 803
 factory, 812
 registered object, 813
 HeaderBlock class, 883, 892
 Heat transfer, 751–752
 HeightField class, 503, 504
 Hierarchical culling, 293–296
 entry point, 294
 logic, 294–295
 See also Culling
 Homogeneous matrices, 40–43
 defined, 40
 determining, 69
 operations, 42–43
 projection, 55
 Homogeneous points, 40–43
 along a line, 42
 concept, 42
 defined, 40
 See also Points
 Homogeneous space. *See* Clip space
 Hooke's law, 516
 Householder transformations, 722
 Hypotenuse, 603

I

Identifier names, 791, 792
 capitalization, 791
 encoding for, 792
 underscores and, 792
 Identifiers, unique, 820–821
 Identity matrix, 674
 Idle loop, 867
 Idle processing, 849
 IKController class, 522
 Image-space algorithm, 375
 Implicit surfaces, 574
 Impostors. *See* Sprites
 Incoming portals, 367
 Incremental garbage collection, 896
 Incremental update matrix, 864
 IndexBuffer class, 222
 Index buffers, 222
 as data members, 253
 defined, 222
 See also Buffers
 Index(es)
 controllers, 225
 recording, 385–386
 Inertia tensor
 computing, 527–528
 rigid bodies, 523, 527–528
 world coordinates, 523
 Inexact culling test, 66–67
 Infinite cylinders, 537
 Infinite level of detail, 378, 387
 Inheritance, 790
 graph, 799
 multiple, 790, 797–799
 single, 793–797
 Initialization
 application layer, 831
 dynamic, 825
 function, 824
 multiple, 825
 pre-main, 822, 839
 registered functions, 839
 static data, 830
 InitializeCameraMotion
 function, 853
 InitializeFactory function, 812
 Initialize function, 826–827, 841, 842
 InitializeObjectMotion
 function, 857
 Initial value problem, 747
 Inner-pointing, 45, 47
 Insertion sort, 468–469
 Instances, 230
 Instancing, 230, 232
 Integer-valued coordinates, 153
 Integration, 742–747
 Gaussian quadrature, 544, 746–747
 Romberg, 543–544, 742–746
 See also Numerical methods
 Integrity, software, 784
 Interpolation
 exact, 548
 Hermite basis, 562
 of orientation, 318
 of position, 317–318
 quaternions, 774
 queries, 317
 rotation matrices, 763
 of scale, 318–320
 spherical linear, 763
 Intersection methods, 390, 681–717
 culling objects by planes, 710–717
 line and capsule, 703–708
 line and cone, 710
 line and cylinder, 710
 line and ellipsoid, 709
 line and lozenge, 708–709
 line and OBB, 686–688
 line and plane, 657
 line and quadric surface, 709–710
 line and sphere, 698–700
 line and sphere-swept volume, 703–709
 linear component and convex objects, 681–684
 linear component and oriented box, 686–698
 linear component and planar component, 684–686
 linear component and sphere, 698–703
 ray and OBB, 688–691

- Intersection methods (*continued*)
 - ray and sphere, 700–701
 - segment and OBB, 691–693
 - segment and sphere, 701–703
- Intersections
 - edge, 495–497
 - fast, sort-and-sweep for, 497
 - points, 651
 - self, 521
- Intervals
 - active, 466
 - configurations, 445
 - initially overlapping, 448
 - initially separated, 446–448
 - intersecting, 466–471
 - moving, contact between, 446–448
 - moving apart, 446
 - moving toward each other, 447
 - notation, 106
 - overlapping, 469, 497
 - radius, 616
 - separated, 445
 - sorted endpoints, 466
- Inverse function, 242
- Inverse kinematics, 339–347
 - numerical solution by
 - cyclic coordinate descent, 342–346
 - numerical solution by
 - Jacobian methods, 341–342
 - numerical solution by
 - nonlinear optimization, 342
 - problem, 339, 340
 - variations and, 340
- Inverse mapping, 38
- Inverse square root, 754–755
- Inverse tangent, 756–757
- Inverse transformation, 241–244
 - computation, 241
 - implementation, 243–244
- Iridescence, 951–954
 - application screenshots, 954
 - defined, 951
 - interpolation factor, 953
 - pixel program, 953
 - shader programs, 951–954
 - vertex program, 952
- IridescenceEffect class, 212, 213
- IsValidFrame function, 265
- IsVisible function, 292
- J**
- Jacobian matrix. *See* Derivative matrix
- Jacobian methods, numerical solution, 341–342
- Joints
 - parameter restriction, 346
 - prismatic, 340
 - processing order, 346
 - revolute, 340
- K**
- Keyframe animation, 317–320
 - data use, 316
 - interpolation of orientation, 318
 - interpolation of position, 317–318
 - interpolation of scale, 318–320
 - potential problem, 316
- Keyframe compression, 316, 320–339
- KeyframeController class, 318
- Keyframes
 - controller data, 309
 - controllers, 350
 - defined, 315
 - orientation, 320
 - position, 319
 - samples, 321
 - transformed values as, 321
- Key presses, 847
- Kinematics
 - defined, 339–347
 - forward, 339
 - inverse, 339–347
- Knots, 322
 - in bracketed portion, 326
 - computing, 330–331
 - consecutive, difference, 337
 - distinctive, number of, 331
 - division, 331
 - nonuniform, 322
 - uniform, 322
- Knot vectors
 - automatic generation, 559
 - control point modification and, 560
 - defined, 322
 - nonuniform, 552, 559
 - open, 322, 552
 - periodic, 322, 552
 - rows, 553
 - types of, 552–553
 - uniform, 552
- L**
- Leaf nodes, 228, 273
 - BSP tree, 364
 - render state at, 282
- Leaks, memory, 876
- Least-squares error function, 323
- Least-squares fitting, 628, 724–732
 - circle to 2D points, 727–729
 - linear, of points, 724
 - linear, of points with

- orthogonal regression, 725–726
 - planar, of points, 726
 - planar, of points with orthogonal regression, 726–727
 - quadratic curve to 2D points, 731
 - quadric surface to 3D points, 731–732
 - sphere to 3D points, 729–731
 - See also* Numerical methods
- Least-squares minimization, 727
- Left-handed coordinate systems, 929
 - defined, 10
 - determination, 13
 - for faces of cubes, 930
 - geometric illustration, 12
 - See also* Coordinate systems
- Left-hand rule, 10
- Level class, 503, 504
- Level-connector multigraphs, 483
- Level of detail (LOD)
 - continuous, 378, 380–387
 - discrete, 378, 379–380
 - geometric, 377
 - infinite, 378, 387
 - nodes, 379
- Levels
 - connectors, 487–488
 - defined, 482
 - moving between, 486–488
 - multiple connections
 - between, 488
 - sets of, 482
 - transporter, 488
- Light class, 262–266
 - hierarchy, 263
 - interface, 264
 - light types support, 263
- Lighting, 92–99, 261
 - ambient, 95
 - Blinn, 97
 - categories, 95
 - defined, 94
 - diffuse, 95–96
 - dynamic, 92
 - equation, 98–99
 - high dynamic range, 95
 - models, 95
 - Phong model, 96
 - specular, 96–97
 - tangent-space, 916–919
 - vertex shader programs, 303
- Lighting application, 899, 906
- Lights
 - ambient, 93, 262, 303, 901
 - colors, 264
 - concept support, 261–262
 - creating, 264
 - directional, 93, 97, 303, 902–903
 - geometric primitives and, 199
 - point, 93, 903–904
 - sources, 93–94
 - Spatial class, 265
 - spotlights, 93, 97, 264, 904–909
 - Wild Magic implementation, 94
- Linear complementarity problem (LCP), 527
- Linear components, 529–532
 - convex object intersection, 681–684
 - defined, 529
 - to linear component, 642–646
 - to oriented box, 663–667
 - oriented box intersection, 686–698
 - planar component
 - intersection, 684–686
 - point to, 639–641
 - to rectangle, 657–661
 - spheres and, 698–703
 - to triangle, 651–655
 - See also* Lines; Rays; Segments
- Linear filtering, 107
- Linear fog, 123
- Linear interpolation, 104, 125
- Linear-linear filtering, 111, 112
- Linear momentum, 522–523
- Linear-nearest filtering, 111
- Linear systems, 719–720
- Linear transformations, 18–28
 - block-matrix form, 40
 - characterization, 43
 - defined, 18
 - examples, 18–19
 - form, 19
 - reflection, 23–24
 - rotation, 20–23
 - scaling, 24–26
 - shearing, 27–28
 - use, 19
 - See also* Transformations
- Line-box find-intersection query, 693–698
- Line loop, 252
- Line-object query, 682–683
- Line of anisotropy, 115
- Lines
 - capsule intersection, 703–707
 - cone intersection, 710
 - cylinder intersection, 710
 - data structure, 654
 - defined, 529
 - drawing algorithm, 78, 82, 148
 - ellipsoid intersection, 709
 - homogeneous points along, 42
 - least-squares, 634–635, 725
 - to lines, 642–643

Lines (continued)

- lozenge intersection, 708–709
- not parallel to capsule axis, 706–707
- to oriented boxes, 664–666
- orthogonal projection onto, 31–32
- parallel to capsule axis, 705–706
- point to, 640
- project to lines, 36–38
- project to points, 665–666
- to rays, 643–644
- to rectangles, 657–659
- rotate to, 343
- to segments, 644
- separating, 393
- slide to, 345
- sphere intersection, 698–700
- to triangles, 651–654
- See also* Linear components

Lines and OBBs, 686–688

- illustrated, 687
- Minkowski difference, 686

Link IDs, 817

LoadBuffer function, 214

Load function, 210, 212, 817

- defined, 204
- implementation, 818
- parsing code, 205

LoadPrograms function, 201

Local control, 322, 558

Local effects, 266

- defined, 196
- lights and, 222
- See also* Global effects

Local heaps, 875

LookDown function, 854

LookUp function, 854

Lozenges, 631–633

- capsule pseudodistance, 462–463
- containing points, 631–633

- defined, 539
- fitting points using mean and covariance, 631–632
- line intersection, 708–709
- lozenge pseudodistance, 463
- medial set, 539
- merging, 633
- minimization method, 632–633
- object culling by planes, 713
- points in, 631
- radius, 708
- rectangle, 708
- sphere pseudodistance, 462

M

Macros

- RTTI support, 799
- typesafety, 801

Main class, 825

Main macros, 799

Manifold meshes, 503

Manipulator

- defined, 339
- illustrated, 339
- joints, 346
- list, with multiple end effectors, 345–346
- list, with one end effector, 342–345
- tree, 346

Mark-and-sweep methods, 896

Masses

- accessing, 515, 519
- boundary, 518
- curve, 510–513
- differential, 622
- indexed, 518
- matrix, 523
- surface, 513–516
- total of all edges, 622
- triangle, 622, 623
- volume, 516–519

MassSpringArbitrary class, 520

MassSpringCurve class, 511, 514–515

Mass-spring systems, 510–521

- arbitrary configurations, 519–521
- curve masses, 510–513
- number of particles, 512
- surface masses, 513–516
- volume masses, 516–519

MassSpringVolume class, 517–518, 519

Matches function, 215

Material.cg, 899

Materials

- concept support, 261–262
- object, 94
- shader program, 899

Matrix2 class, 130–131

Matrix3 class, 525, 777

Matrix4 class, 132

Matrix (matrices)

- banded, 324
- column-major order, 131
- companion, 738
- composition, 134
- diagonal scaling, 861
- identity, 674
- incremental update, 864
- inversion, 236, 324
- mass, 523
- nonuniform scaling, 235
- orthographic, 138–139
- orthonormal, 674
- projection, 136–139, 181
- reflection, 135, 181
- representation, 129–133
- rotation, 759–763
- row-major order, 131
- shear, 779
- storage, 129–133
- symmetric, 324
- vector application, 129, 130
- vector-on-left, 133

- vector-on-right, 133
- view, 134–135
- Matrix–vector product, 130
- Medial sets
 - defined, 538
 - lozenges, 539
- Memory
 - AGP, 174, 183, 194
 - allocation with buddy-system methods, 891–894
 - allocation with segregated-storage methods, 895
 - allocation with sequential-fit methods, 882–891
 - automatic management, 790
 - blocks, maximum size, 881
 - bounded, 896
 - budgets for game consoles, 873–875
 - compaction, 882, 895–896
 - eternally fragmented, 882
 - hierarchy, 194
 - leak detection, 875–882
 - leaks, 896
 - management, 873–896
 - management concepts, 882–896
 - maximum, user specification, 881
 - overruns, 881
 - reclamation, 882
 - speed of allocation/deallocation, 882
 - statistics, 881
 - utilization, 882
 - VRAM, 174, 183
- Memory blocks
 - defined, 882
 - free list, 887
 - power-of-two, 894
 - used, coalescing, 889, 890
- Memory class, 876
- Memory manager, 876
 - binary buddy method, 894
 - statistics, 881
- MemoryManager class, 874
 - advantage, 875
 - local heaps, 875
- Merging
 - bounding volumes, 251
 - boxes, 625–627
 - capsules, 629–630
 - cylinders, 635–636
 - ellipsoids, 638
 - lozenges, 633
 - multiple boxes, 627
 - spheres, 616–617
- Meshes
 - edges, 621–622
 - faces, 622–623
 - solid, 624
 - triangle, 99, 256–258
- Method of rotating calipers, 624
- Method of separating axes, 391, 393–443
 - convex polygons, 407–409
 - convex polyhedra, 410–412
 - extrema of convex polygons/polyhedra, 394–404
 - objects moving with constant linear velocity, 412–436
 - oriented bounding boxes, 436–443
 - point-in-box queries, 671
 - stationary objects, 404–412
- Microsoft Xbox 360, 1
- Midpoint algorithm, 160
- Midpoint distance subdivision, 567–568, 569
- Midpoint line algorithm, 81–82
- Midpoint method, 748
- Miles Sound System, 870
- Minimization, 732–736
 - Brent's method, 732–733
 - conjugate gradient search, 734–735
 - methods in many dimensions, 733–736
 - methods in one dimension, 732–733
 - Powell's direction set method, 735–736
 - steepest descent search, 733–734
 - See also* Numerical methods
- MinimizeOn function, 736
- Minimum-estimate vertex, 493
- MinimumSphere3 function, 613, 614
- Minimum-volume boxes, 624–625
- Minimum-volume ellipsoids, 637
- Minimum-volume sphere, 611–615
 - computation, 611
 - three supporting points, 613
 - two supporting points, 613
- Minkowski different
 - lines and OBBs, 686
 - rays and OBBs, 688
 - segments and OBBs, 691
- Mipmapping, 106, 108–116
 - defined, 108
 - selection, 116
 - Wild Magic implementation, 116
- Mirror-repeated coordinates, 101, 102
- Model space, 8
 - defined, 48
 - object in, 49
 - points, 50
- Modifiers, 226
- Modular continuity, 269
- Modularity, 785–787
- Modules
 - commonality within subgroups, 788
 - criteria, 785–786

- Modules (*continued*)
 - defined, 785
 - object-based, 789
 - open-closed principle, 787
 - related routines, 788
 - representation independence, 788
 - reusability and, 788
 - variation in types, 788
 - MorphController class, 349, 521
 - Morphing
 - defined, 349
 - vertex, 316, 349–350
 - Morrowind, 306
 - Motion equations, 507
 - Mouse events, 848
 - MoveBackward function, 853
 - MoveCamera function, 855, 856, 868, 871
 - MoveDown function, 853
 - MoveForward function, 480, 853, 856
 - MoveObject function, 858
 - MoveScene function, 868
 - MoveUp function, 853
 - Moving objects, 390
 - with constant linear velocity, 412–436
 - finding collisions between, 444–455
 - GetExtremeIndex function and, 419
 - Multigraphs, 482
 - level-connector, 483
 - room-doorway, 482–483
 - Multi-Media Extensions (MMX), 143
 - Multipass drawing, 300, 302–304
 - code block, 302–303
 - ShaderEffect support, 302
 - single-pass, 302–304
 - Multipass effects, 195
 - Effect object and, 195
 - vertex attributes, 188
 - Multipass rendering, 906
 - Multiple-effect drawing, 304–305
 - Multiple-inheritance systems, 797–799
 - defined, 797
 - directed graph, 799
 - hierarchy system, 797
 - See also* Run-time type information (RTTI)
 - MultitextureEffect class, 911
 - Multitextures, 116–117, 911–913
 - added during rendering, 913
 - blended, 913
 - defined, 116
 - drawing with, 300
 - dynamic, 116–117
 - multiplied during rendering, 912
 - shader program, 911–913
 - static, 116
 - TextureEffect object and, 300
 - See also* Textures
 - MyConsoleApplication class, 841
 - MyFunction class, 808
 - MyWindowApplication class, 843–844
- N**
- Name strings, 820
 - Naming conventions, 791
 - Narrow phase, collision detection, 465
 - Natural splines, 549, 550
 - Nearest-linear filtering, 111
 - Nearest-nearest filtering, 110
 - Nearest-neighbor filtering, 105
 - Near plane
 - defined, 43
 - distance, 45
 - See also* Planes
 - NetImmerse, 2, 306
 - Networking engine, 870
 - Newton's iteration scheme, 673
 - Newton's method, 448, 449, 452
 - in many dimensions, 741
 - in one dimension, 737
 - Node-based sorting, 365–366
 - Node class, 221, 223–225, 806
 - member functions for
 - geometric updates, 272
 - objects, 309, 312
 - in render-state updates, 280–281
 - Nodes
 - BSP, 358, 364
 - child, 229–230, 270
 - grouping, 227
 - leaf, 228, 273, 282
 - list, transverse, 287
 - LOD, 379
 - subtree, 275
 - tagging, 311
 - top-level, 312
 - tree, 230
 - NoIntersect function, 419–421, 428, 430
 - Nonlinear optimization, 342
 - Nonuniform knots, 322
 - Nonuniform rational B-spline.
 - See* NURBS curves
 - Nonuniform scaling, 319
 - curse, 778–782
 - defined, 25
 - diagonal matrix, 235
 - problems, 319
 - of sphere, 239
 - Wild Magic and, 234, 236
 - See also* Scaling
 - Nonuniform subdivision
 - lengths of nonlinear terms, 596
 - rectangle patches, 594–596

- triangle patch subdivision, 603–607
 - See also* Subdivision
 - Normalization cube map, 923
 - Normalization maps, 95
 - Normalized device coordinates (NDCs), 56
 - Normalized time, 317, 318
 - Normalized viewpoint
 - coordinates, 473
 - Normalize function, 654
 - Normal maps
 - construction implementation, 916
 - defined, 914
 - generating, 914–916
 - illustrated, 917
 - NumericalConstant class, 207, 209
 - Numerical constants, 192, 207
 - Numerical methods, 719–757
 - differential equations, 747–754
 - eigensystems, 722–724
 - fast function evaluation, 754–757
 - integration, 742–747
 - least-squares fitting, 724–732
 - minimization, 732–736
 - root finding, 736–742
 - systems of equations, 719–722
 - Numerical Recipes in C*, 719
 - NURBSCurve class, 561–562
 - NURBS curves, 542, 560–562
 - control points, 560
 - defined, 560
 - encapsulation, 561
 - evaluator, 561–562
 - flexibility, 560
 - general parameterization, 560–561
 - See also* Curves
 - NURBS rectangle patches, 583–584
 - control points, 583
 - defined, 583
 - NURBSSurface class, 583–584
 - Nyquist frequency, 324
- O**
- Object class, 228, 317, 802
 - API, 812–819
 - classes derived from, 810
 - name strings and, 820
 - static unsigned integer member, 821
 - unique identifier support, 821
 - Object culling, 66–67
 - by planes, 710–717
 - capsules, 712–713
 - cones, 716–717
 - convex polygons, 717
 - convex polyhedra, 717
 - cylinders, 715–716
 - defined, 66, 710
 - ellipsoids, 713–715
 - exact culling test, 66
 - inexact culling test, 66–67
 - lozenges, 713
 - oriented boxes, 711–712
 - spheres, 712
 - as test-intersection query, 711
 - See also* Culling
 - Object-oriented infrastructure, 783–872
 - application layer, 831–872
 - initialization/termination, 822–830
 - names, 819–820
 - namespaces, 790–793
 - naming conventions, 790–793
 - run-time type information, 793–799
 - software construction, 783–790
 - streaming, 808–819
 - style, 790–793
 - templates, 800–801
 - unique identifiers, 820–821
 - Object picking, 472–480
 - pick ray construction, 472–474
 - scene graph support, 475–479
 - Objects
 - application, 839
 - boxes, 534–535
 - convex, 391, 392
 - coordinate system, 857
 - culling, 245
 - curve mass, 511
 - describing, 789
 - drawing, 179–180
 - drawing algorithm, 217
 - dynamic, 392
 - finding, 789
 - grouping together, 224
 - linear component, 529–532
 - materials, 94
 - model bound association, 224
 - motion, 857–865
 - moving, 390
 - moving with constant linear velocity, 412–436
 - orientation on curved paths, 570–571
 - planar component, 532–534
 - polygonal, 485–486
 - portal-system, 312
 - quadrics, 535–538
 - registered, 813
 - related, hierarchy, 223
 - render states, 219
 - shared, 230–233, 802–808

- Objects (*continued*)
 - sorting, 221
 - sphere-swept volumes, 538–539
 - standard, 529–539
 - static, 392
 - stationary, 390, 404–412
 - 3D, 7
 - top-level, 809, 815
- Object space, 48–50
 - algorithm, 376
 - defined, 48
 - See also* Model space
- ObjectSystem folder, 799
- Oblique projection, 33–34
- Obstacle class, 503, 504
- Obstacles, 486, 503
- Occluders, 376
- Occlusion culling, 368, 375–376
 - depth buffering as, 375
 - dynamic, 376
 - occluders, 376
 - See also* Culling
- Occlusion queries, 166
- Offscreen rendering, 172
- OnDisableTexture function, 190
- OnEnableTexture function, 189–190
- OnFrameChange function, 177
- OnFrustumChange function, 177
- OnIdle callback, 849, 865, 867–868
 - loop, 872
 - physics engine, 869
 - for single-threaded application, 867
- OnInitialize function, 203
- OnLoadProgram function, 210–211, 212, 213, 969
- OnLoadVProgram function, 186
- OnReleaseProgram function, 187, 212
- OnSpecialKeyDown callback, 847, 871
- OnViewportChange function, 139, 140
- Open-closed principle, 226, 787
- OpenGL, 7, 125
 - anisotropic filtering extension, 115–116
 - clipping planes, 183
 - color data packaging, 214
 - conventions, 128–145
 - fast computations, 143–144
 - matrix composition, 134
- OnLoadVProgram
 - implementation, 186
- projection matrix, 136–137
- renderer, 65, 148
- right-handed window
 - coordinates, 141
 - rotations, 141–142
- selection mechanism, 114
- vector/matrix conventions, 8
- vector-on-right, column-major order, 133
- view matrices, 135
- window handedness, 139
- Open knot vectors, 322
- Optimal fit policy, 890
- Order dependencies, 825
- Ordinary differential equations, 747–750
 - Euler’s method, 748
 - first-order system, 747
 - midpoint method, 748
 - Runge-Kutta fourth-order method, 748–749
 - Runge-Kutta with adaptive step, 749–750
 - second-order system, 747
 - See also* Differential equations
- Organization, this book, 3–5
- Orientations
 - with Frenet frame, 571
 - interpolation of, 318–319
 - keyframes, 320
- objects on curved paths, 570–571
- Oriented bounding boxes (OBBs), 244, 436–443
 - with center point, 437
 - defined, 436, 535
 - face normals, 437, 439
 - intersection testing, 437
 - lines and, 686–688
 - potential separating directions, 440
 - projection intervals, 438
 - rays and, 688–691
 - segments and, 691–693
 - symmetry, 437, 438
- Oriented boxes
 - linear component intersection, 686–693
 - linear components to, 663–667
 - lines to, 664–666
 - object culling by planes, 711–712
 - to oriented boxes, 670–672
 - points to, 663
 - projection onto plane perpendicular, 665
 - projection to rectangle, 665
 - rays to, 666
 - rectangles to, 669–670
 - segments to, 666–667
 - triangles to, 667–668
 - See also* Boxes
- Orthogonal frustum, 60
- Orthogonal matrices, 24
- Orthogonal projection
 - analysis, 31
 - onto line, 31–32
 - onto plane, 32–33
- Orthogonal regression
 - linear fitting of points with, 725–726
 - planar fitting of points with, 726–727

Orthographic matrix, 138–139
 Outdoor environment, moving
 in, 488
 Outgoing portals, 367, 372
 Outside region, 501
 Overlapping intervals, 469
 Overlapping rectangles, 471
 Overshooting, 563

P

Parameterized data types. *See*
 Templates
 Parametric curves, 542
 Parametric subdivision
 by arc length, 566–567
 by midpoint distance,
 567–568
 by uniform sampling, 566
 for cubic curves, 568–570
 curves, 566–570
 rectangle patches, 587–602
 surfaces, 587–607
 triangle patches, 602–607
 See also Subdivision
 Parametric surface patch, 573
 Parent-child relationship, 223
 Parsers, 205
 Parsing
 command line, 834
 shader programs, 201–213
 Partial derivatives, 918, 919
 Partial differential equations,
 750–754
 elliptic, 753
 extension to higher
 dimensions, 753–754
 hyperbolic, 752
 parabolic, 751–752
 second-order, 751
 use, 750
 See also Differential
 equations
 Partial subdivision, 598–599
 defined, 597
 with one subdividing edge,
 601
 parent's topological
 constraint, 599, 600
 with three subdividing edges,
 599
 with two adjacent
 subdividing edges, 600
 with two opposing
 subdividing edges, 600
 See also Subdivision
 ParticleController class, 521
 Particles
 defined, 258
 equation of motion, 513
 with finite mass, 510
 in mass-spring system, 512
 as rigid bodies, 522
 Particles class, 226, 258
 ParticleSystem class, 509, 510,
 511
 Particle systems, 316, 508–510
 defined, 258
 physical simulation of, 508
 Partitioning of *st*-plane
 by rectangle domain, 656
 by triangle domain, 647
 PassThrough.cg, 898
 Path controlling, 570
 Pathfinding, 481–506
 algorithm, 485, 486
 blueprints, 488
 data structures, 503–504
 defined, 481
 envelope construction,
 494–503
 environments, 482
 implementation in 3D, 489
 levels, 482
 moving between levels,
 486–488
 moving between rooms, 486
 moving through outdoor
 environment, 488
 reachability, 481
 rooms, 482–486
 visibility, 481
 visibility graph calculation,
 504–506
 visibility graphs, 489–494
 Paths
 between adjacent rooms, 487
 compound, 488
 connecting rooms, 486
 connecting two points, 487
 destination location, 481
 movement, 485
 polyline, 484
 shortest, 485
 source location, 481
 Performance
 portability versus, 127–128
 quaternions, 777–778
 repackaging and, 310
 Periodic knot vectors, 322
 Perspective camera model,
 43–48
 PerspectiveInterpolate
 function, 125
 Perspective interpolation, 125
 Perspective projection
 conics project to conics, 39
 lines project to lines, 36–38
 onto plane, 34–35
 properties, 35–39
 triangles project to triangles,
 38–39
 Phong lighting model, 96
 Phong shading, 94–95
 Physics, 507–528
 Physics engines, 316, 507
 deformable bodies, 521–522
 managing, 869–870
 mass-spring systems,
 510–521
 particle systems, 508–510

- Physics engines (*continued*)
 - rigid bodies, 522–528
- Picking, 472–480
 - applications, 478–479
 - defined, 472
 - hierarchical, 475
 - pick ray construction, 472–474
 - triangle mesh level and, 478
- PickRecord class, 475–476
- Pitch, 858
- PixelProgram class, 205
- Pixels
 - circle, 82
 - cracking, 82
 - pipeline operations, 166–167
 - selection, 78
 - shared, 163
 - span endpoints, 124
 - surface attributes, 92
 - vertex attributes, 92–125
- Pixel shader programs, 91, 164–167, 299
 - applying, 164
 - calling, 164, 166
 - input, 165
 - parameters, 165
 - software version, 165
 - textures, 967
 - See also* Shader programs
- Placement new operators, 875
- Planar components, 532–534
 - defined, 533
 - linear component
 - intersection, 684–686
 - rectangles, 534
 - triangles, 533–534
- Planar curves, 542
- PlanarReflectionEffect class, 935–936
- Planar reflections, 935–939
 - defined, 935
 - effect, 938
 - handling, 935
 - illustrated, 939
- PlanarShadowEffect class, 939–940
- Planar shadows, 939–942
 - defined, 939
 - illustrated, 942
 - projection matrix, 942
 - shadow caster, 941–942
 - See also* Shadows
- Plane-at-a-time clipping, 71–74, 154
 - copying vertices and, 74
 - defined, 71
 - drawback, 74
 - pseudocode, 71
 - single triangle, pseudocode, 72–74
 - triangle splitting
 - configurations, 72
 - See also* Clipping
- Plane-at-a-time culling, 245, 246
- Plane class, 363
- Planes
 - clipping, 155
 - defined, 532
 - distance to origin, 532
 - far, 43, 45
 - model-space, 242
 - near, 43, 45
 - object culling by, 710–717
 - oblique projection onto, 33–34
 - origin, 532
 - orthogonal projection onto, 32–33
 - perspective projection onto, 34–35
 - rotate to, 343–344
 - slide to, 345
 - view, 43
- Platforms, 310
- PointController class, 350, 521
- Pointers
 - controller, 815
 - function, 826
 - parent, 815
 - smart, 288–289, 802, 806–807
- Point-in-sphere query, 609–610
- Point lights
 - defined, 93
 - plane and sphere
 - illumination, 907, 909
 - shader program, 903–904
 - See also* Lighting; Lights
- Points, 9
 - affine transformation
 - between, 29–30
 - box containing, 618–625
 - in boxes, 617–618
 - in capsules, 627–628
 - capsules containing, 628–629
 - to circles in 3D, 675–676
 - collection of, 252
 - in cylinders, 634
 - cylinders containing, 634
 - to ellipses, 672–673
 - in ellipsoids, 636–637
 - to ellipsoids, 673–674
 - fitting, with B-spline curves, 321–324
 - fitting circles to, 727–729
 - fitting quadratic curve to, 731
 - fitting quadric surface to, 731–732
 - fitting spheres to, 729–730
 - fitting with mean and covariance, 618–621
 - as 4-tuples, 40
 - homogeneous, 40–43
 - at infinity, 42
 - to linear components, 639–641
 - linear fitting, 724
 - linear fitting, with

- orthogonal regression, 725–726
- to lines, 640
- in lozenges, 631–633
- model space, 50
- oblique projection, 33
- to oriented boxes, 663
- origin, 9
- orthogonal projection, 31, 32
- perspective projection, 35
- planar fitting, 726
- planar fitting, with
 - orthogonal regression, 726–727
- to quadratic curves, 674–675
- to quadric surfaces, 674–675
- to rays, 640–641
- to rectangles, 655–657
- rotate to, 343
- to segments, 641
- shearing of, 27
- slide to, 344–345
- sphere centered at average of, 610–611
- sphere containing, 609–615
- sphere containing axis-aligned box of, 610
- system, 350
- to triangles, 646–651
- vectors and, 15–17
- Point systems, 316
- Polar decomposition, 235–236, 779, 781
- Polygon-of-intersection
 - clipping, 74–77, 154
 - defined, 74
 - implementation, 79–81
 - line pixel decision, 79
 - midpoint line algorithm, 81–82
 - triangle clipped against
 - bottom frustum plane, 75
 - triangle clipped against left frustum plane, 76
 - triangle clipped against right frustum plane, 76
 - triangle clipped against top frustum plane, 75
 - triangle intersecting frustum at multiple faces, 75
 - Wild Magic implementation, 77
 - See also* Clipping
- Polygons
 - boundary, 483
 - bounding, 313
 - edges for containment
 - testing, 401
 - inside frustum, 155
 - list, 357
 - outside frustum, 155
 - portal, 368
 - separated by edge-normal
 - direction, 407
 - spherical, 398, 399
 - subpolygons, 402, 403
 - See also* Convex polygons
- Polyhedra
 - convex, 245, 394–404, 409–410, 427–436
 - spherical dual, 396
- Polyline class, 226
- Polylines, 313
 - closed, 252
 - shared endpoints, 255
- Polymorphism, 790, 793
- Polynomial roots, 737–740
 - bounding by derivative
 - sequences, 738–739
 - bounding by Sturm
 - sequences, 739–740
- Polynomials
 - characteristic, 722, 738
 - coefficients, 546, 578, 579
 - quartic, 672, 673, 721
 - roots, 675
 - Sturm, 740
 - systems, 720–722
- Polypoint class, 226, 252, 253
- Popping effect, 380
- PopState function, 282
- Population dynamics, 751–752
- Portability, software, 784
- Portal class, 296
 - constructor, 372, 375
 - interface, 371–372
 - objects, 312
- Portals, 366–375
 - configuration, 368
 - culling code, 374
 - defined, 366
 - drawing system, 374
 - graph connections between
 - regions and, 367
 - incoming, 367
 - outgoing, 367, 372
 - polygons, 368
 - in sorting, 367
 - update, 372–373
 - Wild Magic system, 367
- Positions
 - interpolation of, 317–318
 - keyframes, 319
- Positive definite, 536
- Positivity, 646
- Post-main function, 822
- Post-main termination, 825
- PostMessage function, 847
- Potential theory, 753
- Powell's direction set method, 735–736
 - defined, 735
 - pseudocode, 735–736
- Pre-main mechanism, 822
- Prismatic joint, 340
- Probing, 481
- Program class, 204–205
 - defined, 204
 - interface, 204–205
 - object, 208–209
- Program text string, 207

ProjectedTextureEffect class,
943–944

Projected textures, 943–945
defined, 99, 943
illustrated, 946
implementation, 945
See also Textures

Projection graphs, 494–495

Projection matrices, 136–139

Projection space. *See* Clip space

Projective geometry, 42

Projective transformations,
31–39
oblique projection onto
plane, 33–34
occurrence, 43
orthogonal projection onto
line, 31–32
orthogonal projection onto
plane, 32–33
perspective projection onto
plane, 34–35
See also Transformations

Projector sources, 182

ProjInfo class, 429

PropagateStateFromRoot
function, 282, 284, 286

Pseudodistance, 444–445
capsule and box, 464
capsule and capsule, 462
capsule and lozenge, 462–463
capsule and triangle, 464
convex polygons and convex
polyhedra, 465
derivative, estimating, 461
functions, 391, 444
lozenge and lozenge, 463
for specific pairs of object
types, 461–465
sphere and box, 463
sphere and capsule, 462
sphere and lozenge, 462
sphere and sphere, 461
sphere and triangle, 464

sphere-swept volumes,
461–463
time-varying, 450–452

PushState function, 282

Q

Quadratic curves
fitting to 2D points, 731
points to, 674–675

Quadratic equations
discriminant, 698
for root finding, 706, 707
solving two in two
unknowns, 721

Quadratic error metrics,
380–385
construction, 383
simplification, 384–385
surface attribute selection,
385
topological considerations,
383–384

Quadratic forms
constrained, extrema of,
723–724
defined, 722
extrema of, 722–723

Quadratics, 535–538
cones, 537–538
cylinders, 537
ellipsoids, 535
spheres, 535

Quadric surfaces
defined, 574
fitting to 3D points, 731–732
line intersection, 709–710
points to, 674–675

Quaternion class, 525, 777

Quaternions, 764–774
to axis-angle, 773
axis-angle to, 772
classical view, 770–772
interpolation, 774

linear algebraic view,
766–769
to matrix, 773
matrix to, 773
memory usage, 777
multiplication of, 771
representation, 771–772
unit, 772

Queries
find-intersection, 681–684,
685, 693–698
line-object, 682–683
multiple distance, 659
point-in-box, 671
ray-object, 682–683
segment-object, 683–684
test-intersection, 681,
686–693

R

Range-based fog, 123

Rasterization, 77–92, 158–159
circles, 82–83
defined, 77
ellipses, 84–89
line segments, 77–82
with top-left rule, 163
triangles, 89–92
vertex attributes, 124–125

RasterizeEdge function, 125,
164

Rasterizers, 159

RasterizeTriangle function,
91, 125, 158, 161

Rational arithmetic, 495–497

RationalArithmetic directory,
496

Ray-object query, 682–683

Rays
defined, 529
lines to, 643–644
to oriented boxes, 666
point to, 640–641

- to rays, 645
- to rectangles, 659–660
- to segments, 645
- sphere intersection, 700–701
- to triangles, 654
- See also* Linear components
- Rays and OBBs, 688–691
 - illustrated, 689
 - Minkowski difference, 688
 - projections, 689
 - test-intersection query
 - implementation, 689–691
- Rectangle patches
 - Bézier, 574–578
 - B-spline, 582–583
 - defined, 573
 - NURBS, 583–584
- Rectangle patch subdivision, 587–602
 - camera model adjustments, 596–597
 - cracking, 597–602
 - nonuniform subdivision, 594–596
 - number of vertices, 590
 - parameter space, 588
 - recursive, 589, 594
 - uniform subdivision, 587–594
 - See also* Parametric subdivision
- Rectangles, 534
 - axis-aligned, 471
 - defined, 534
 - intersecting, 471
 - linear component to, 657–661
 - lines to, 657–659
 - to oriented boxes, 669–670
 - overlapping, 471
 - points to, 655–657
 - rays to, 659–660
 - to rectangles, 661–663
 - rotation, 624
 - segments to, 660–661
 - to triangles, 661–663
- Recursive subdivision, 570
- Red Orb, 306
- Reference counting, 802–808
 - defined, 802
 - smart pointers, 805–807
- Reflection, 23–24
 - determinant value, 24
 - matrices, 24, 135, 181
 - planar, 935–939
 - of vector through plane, 23
 - See also* Linear transformations
- Refraction, 932–935
 - angles of incidence and, 933
 - defined, 932
 - of light beam, 932
 - light direction, 933
 - observation, 934
 - shader programs, 934–935
- Registered objects, 813
- RegisterFactory function, 812
- Register function, 813
- Relaxation methods, 748
- Renderer class, 138, 266
 - as abstract, 175
 - resource management system, 183
 - screen dimensions, 474
- RendererConstant class, 207, 209
- Renderer constants, 191–192, 207
- Renderer function, 266
- Renderers, 147–216
 - derived-class, 182
 - Direct3D, 65, 148
 - heart of, 194–216
 - maximum number of colors, 182
 - OpenGL, 65, 148
- Rendering
 - abstract API, 175–194
 - double-buffer, 178
 - global states effects, 261
 - hardware, 173–175
 - offscreen, 172
 - software, 149–173
- Render-state coherency, 219
- Render states, 259–268
 - effects, 266–268
 - global state, 259–261
 - at leaf node, 282
 - lights, 261–266
 - organization, 219
- Render-state updates, 280–289
 - defined, 268
 - entry point, 285–286
 - interface portions for, 280–281
 - propagation down tree, 288
 - pushing/popping, 287
 - semantics, 284
 - situation illustration, 282
 - smart pointers, 288–289
 - See also* Updates
- Render targets, 172
- Render-to-texture, 172
- Reparameterization by arc length, 541, 543–544
- Repeated coordinates, 101, 102
- RequestTermination function, 847
- Resource management, 182–194
 - Renderer class, 183
 - system, 174
- Resources
 - catalogs, 193–194
 - enabling/disabling, 189–193
 - loading/releasing, 183–189
- RestorePostWorldTransformation function, 181
- RestoreWorldTransformation function, 181
- Reusability, 784, 787–788
 - defined, 784

- Reusability (*continued*)
 - issues, 788
 - See also* Software
- Revolute joint, 340
- Revolution surfaces, 586
- Richardson extrapolation, 742–744, 745
- Right direction, 8
- Right-handed coordinate systems
 - defined, 10
 - determination, 13
 - geometric illustration, 12
 - See also* Coordinate systems
- Right-hand rule, 10, 11
- Rigid bodies, 522–528
 - angular momentum, 523
 - angular velocity, 524
 - constant quantities, 526
 - defined, 522
 - inertia tensor, 523
 - initializing, 527
 - linear momentum, 522–523
- RigidBody class, 525–527
 - constructor, 526
 - interface, 525–526
- Ripmaps, 114
- RipplingOcean application, 956
- Robustness, 784
- Roll
 - controlling, 858
 - defined, 857–858
- Romberg integration, 543–544, 742–746
 - code, 745–746
 - interval, 745
 - method, 744–746
- Richardson extrapolation, 742–744, 745
- trapezoid approximations, 744–745
- trapezoid rule, 744
- See also* Integration
- Room class, 503, 504
- Room-doorway multigraphs, 482–483
- Rooms
 - blueprint, 484
 - boundary polygon, 483
 - ceilings, 483
 - doorways, 482–483
 - floors, 483
 - moving between, 486
 - obstacles, 484, 486
 - structure, 483–486
 - walls, 483
- Root finding, 736–742
 - bisection, 737, 741
 - methods in many dimensions, 740–741
 - methods in one dimension, 736–740
 - Newton's method, 737, 741
 - polynomial roots, 737–740
 - process, 736
 - See also* Numerical methods
- RotateTrackBall function, 861–864
- Rotation, 20–23, 759–782
 - about coordinate axes, 21
 - about right vector, 852
 - about up vector, 852
 - about world right vector, 854–855
 - about world up vector, 854
 - angle, 762, 859
 - axis, 762, 771, 859
 - axis, in world coordinates, 860
 - axis direction, 761
 - clockwise, 775
 - commutativity, 237
 - coordinate axis, 774
 - counterclockwise, 21
 - determinant value, 24
 - direction, 21
 - double covering, 772
 - Euler angles, 774–777
 - graphics APIs, 140–143
 - half-angle, 766
 - incremental, calculation, 855
 - to line, 343
 - nonuniform scaling and, 778–782
 - object, via keyboard events, 858
 - performance issues, 777–778
 - pitch, 858
 - to plane, 343–344
 - to point, 343
 - product, 769–770
 - quaternions, 764–774
 - rectangles, 624
 - roll, 857–858
 - speed adjustment, 856
 - 3D, about z-axis, 767
 - trackball, 859–860
 - transformation, 135
 - vector, 769
 - world, 861
 - in *xz*-plane, 22
 - yaw, 858
 - y*-axis, 774
 - See also* Linear transformations
- Rotation matrices, 20, 22, 759–763
 - axis-angle to matrix, 760–762
 - factorization, 775
 - identity, 761
 - interpolation, 763
 - matrix to axis-angle, 762–763
 - representation, 20, 759
 - rotation axis direction and, 761
 - skew-symmetric, 761
 - slerp, computing, 763
 - S*-matrices, 23
 - trace, 762
 - world-to-world, 861
 - xy*, 764

- Round-off errors, 483, 495
 - determinant and, 643
 - finding intersections and, 495
 - Row-major order, 131
 - Run function, 840, 842–843
 - Runge-Kutta fourth-order method, 748–749
 - Runge-Kutta solver, 510, 521
 - Run-time type information (RTTI), 176, 476, 793–799
 - data type, 707, 793
 - defined, 793
 - macro support, 799
 - multiple-inheritance systems, 797–799
 - names, 814, 815
 - single-inheritance systems, 793–797
 - strings, 816
- S**
- SamplerInformation class, 209
 - Save class, 815
 - Save function, 810
 - Scales, interpolation of, 318–320
 - Scaling, 24–26
 - of covariance matrix, 623
 - defined, 24
 - equation, 26
 - illustrated, 25
 - matrix representation, 24–25
 - nonuniform, 25, 234, 236, 319
 - in three dimensions, 26
 - uniform, 25
 - See also* Linear transformations
 - Scan line processing, 161–164
 - loops, 161
 - top-left rule and, 163
 - Scene graph compilers, 128, 305–313
 - compilation semantics, 311–313
 - culling of objects as, 311
 - Scene graph management, 7
 - core classes, 221–226
 - Wild Magic, 220
 - Scene graphs, 217–313
 - as abstract directed graph, 809
 - components, loading, 308
 - defined, 217, 305
 - depth-first traversal, 296
 - design issues, 217–233
 - developmental view, 307
 - drawing, 195–197
 - dynamic, 307
 - examples, 307–308
 - as expressions, 307–311
 - geometric primitives, 196
 - illustrated example, 231, 233
 - levels, 482
 - loading, 816–819
 - picking support, 475–479
 - saving, 812–816
 - semantics, 311
 - static, 307
 - structure, 308–309
 - as top-level data structure, 195
 - updates, 268–269
 - Scissor test, 166
 - Screen matrix, 60
 - SearchByPolicy function, 889
 - Segment-object query, 683–684
 - Segments, 254–256
 - collection, 254
 - defined, 529
 - end points, 254
 - intersection calculation, 436
 - lines to, 644
 - to oriented boxes, 666–667
 - pixel selection based on slope, 78
 - pixels forming best, 78
 - point to, 641
 - rasterization, 77–82
 - rays to, 645
 - to rectangles, 660–661
 - to segments, 645–646
 - sphere intersection, 701–703
 - to triangles, 654–655
 - See also* Linear components
 - Segments and OBBs, 691–693
 - illustrated, 692
 - Minkowski difference, 691
 - projections, 692
 - test-intersection query implementation, 691–693
 - Segregated-storage methods, 895
 - SelectPartitionPlane function, 357
 - Self-intersections, 521
 - Semantics, scene graph, 311
 - Sentinels, 196
 - Separated intervals, 445
 - Separating axes, 393
 - Separating direction, 394
 - Separation test, 444
 - Sequential fit methods, 882–891
 - SetActiveQuantity function, 253
 - SetCamera function, 177
 - SetChild function, 230
 - SetColorBuffer function, 165, 170, 171
 - SetColorMask function, 181
 - SetDepthRange function, 181
 - SetFrustum function, 294
 - SetGlobalState function, 178, 198, 945
 - SetIndexQuantity function, 253
 - SetInterpolate function, 213
 - SetLight function, 178
 - SetLocal function, 270

- SetMatrix function, 239
- SetName function, 820
- SetPostWorldTransformation
 - function, 181
- SetProjector function, 182
- SetScale function, 239
- SetSpring function, 521
- SetTranslate function, 238
- SetViewport function, 139, 473
- SetWorldTransformation
 - function, 181, 199
- Shader-based pipeline, 2
- Shader class, 207–208
 - multiple instances, 208
 - user-defined constant storage, 207
- Shader constants, 175
- ShaderEffect class, 266, 957
 - classes derived from, 213
 - multipass drawing support, 302
 - objects, 209, 967
- Shader effect system, 2
- Shader programs
 - ambient lights, 901
 - attributes, 206
 - bump maps, 919–923
 - Cg, 164, 212
 - “compiled,” 959
 - complexity, 897
 - constants, 191–192
 - constants, dynamic updates, 970
 - creating, in Wild Magic, 957–971
 - cube maps, 929–932
 - default, 215
 - directional lights, 902–903
 - effect, classless, 965–968
 - enabling, 190
 - gloss maps, 923–926
 - for illustrative application, 958–963
 - iridescence, 951–954
 - lighting equation, 98
 - loading, 201–213
 - material, 899
 - multitextures, 911–913
 - numerical constants, 207
 - parsing, 210–213
 - pixel, 164–167
 - point lights, 903–904
 - refraction, 934–935
 - renderer constants, 207
 - skinning, 950–951
 - special effects with, 897–956
 - sphere maps, 926–929
 - spotlights, 904–905
 - stitching, 174, 302, 304, 911
 - textures, 909–910
 - texture samplers, 208
 - user-defined constants, 207–208
 - validation of, 213–216
 - vertex, 68, 97, 149–151
 - vertex colors, 897–899
 - volumetric fog, 948–950
 - water effects, 955–956
- Shader stitching
 - defined, 174, 911
 - with LightingEffect::Configure, 304
 - with MultitextureEffect::Configure, 302
 - See also* Shader programs
- Shading
 - defined, 94
 - flat, 94
 - Gouraud, 94
 - Phong, 94–95
- Shadow maps, 945–947
 - defined, 945
 - depth texture for, 172, 946
- Shadows
 - caster, 941
 - cloud, 957–958
 - generation, 195–196
 - planar, 939–942
 - projection matrix, 941
- Shared objects, 230–233
 - bookkeeping, 802
 - reference counting and, 802–808
- Shearing, 27–28
 - in arbitrary plane, 28
 - direction, 28
 - matrix representation, 27
 - of points, 27
 - three dimensions, 27
 - two dimensions, 27
 - See also* Linear transformations
- Shear matrix, 779
- Shininess, 94
- Shooting methods, 748
- Silhouette edges, 503
- Similarity transformation, 861
- Sine, 755–756
- Single-effect, multipass
 - drawing, 302–304
- Single-inheritance systems, 793–797
 - defined, 793
 - hierarchy illustration, 794
 - root class, 799
- Single-pass drawing, 298–302
- Single-sided cones, 537–538, 710
- Singular value decomposition, 235, 236, 237, 779, 781–782
 - defined, 781
 - iterative approach, 782
- 16-byte data alignment, 17
- SkinController class, 522, 950
- Skinning, 316, 347–349
 - application screenshots, 951
 - defined, 347
 - shader programs, 950–951
 - skin representation, 347
 - support, 348
 - weights and offsets for, 348

- Sliding
 - defined, 344
 - to line, 345
 - to plane, 345
 - to point, 344–345
- SmallTalk, 790
- Smart pointers, 802
 - cleanup, 806
 - comparison, 806
 - copying, 288–289
 - destructor, 807
 - as function parameters, 807
 - Geometry class, 289
 - guidelines, 807
 - NodePtr, 806, 808
 - templates, 803
- Soft addition, 119
- SoftFrameBuffer class, 172
- SoftRenderer class, 136, 138, 172
- Software
 - compatibility, 784
 - construction, 783–790
 - correctness, 784
 - ease of use, 784
 - efficiency, 784
 - engineering goal, 784
 - evolution, 2–3
 - extendability, 784
 - integrity, 784
 - maintenance, 784–785
 - modularity, 785–787
 - object orientation, 789–790
 - portability, 784
 - quality, 784–785
 - reusability, 784, 787–788
 - robustness, 784
 - verifiability, 784
- Software rendering, 149–173
 - alpha blending, 170
 - back-face culling, 151–154
 - clipping, 154–158
 - color masking, 171
 - depth buffering, 169–170
 - edge buffers, 159–161
 - frame buffers, 172–173
 - pixel shaders, 164–167
 - rasterizing, 158–159
 - scan line processing, 161–164
 - stencil buffering, 167–168
 - texture sampling, 171–172
 - vertex shaders, 149–151
- Sony Playstation 3, 1
- Sort-and-sweep algorithm, 497
- Sorted culling, 296–297
 - portal system, 297
 - sorting, 296–297
 - See also* Culling
- Sorting
 - binary space partitioning, 354
 - node-based, 365–366
 - objects, 221
 - in sorted culling, 296–297
 - spatial, 353–376
- SoundEmitter class, 870
- Sound engine, 870
- Sound renderer, 870
- Source blending functions, 118
- Source colors, 117, 170
- Space curves, 543
- Spaces
 - Cartesian, 8, 11
 - clip, 8, 52–55, 164, 181
 - defined, 8
 - model, 8, 48–50
 - view, 8, 50–52, 927
 - window, 8, 56–58, 164
 - world, 8, 48–50
- Spatial class, 221, 222–223, 226, 228, 348
 - bounding volume storage, 250
 - Culler class interaction, 293
 - data members for geometric updates, 268–269
 - hierarchical picking subsystem, 475
 - light support, 265
 - member functions for
 - geometric updates, 271
 - in render-state updates, 280–281
 - world transformation, 270
- Spatial coherency, 218, 219
- Spatial hierarchy design, 226–230
- Spatial sorting, 353–376
 - binary space partitioning, 354–364
 - node-based, 365–366
 - portals, 366–375
- Special effects, with shaders, 897–956
- Specular color, 94
- Specular lighting, 96–97
- Sphere maps, 926–929
 - defined, 926
 - shader programs, 926–929
 - texture coordinate mapping, 926
- Spheres, 535, 609–617
 - bounding, 278
 - box pseudodistance, 463
 - capsule pseudodistance, 462
 - centered at average of points, 610–611
 - containing axis-aligned box of points, 610
 - containing points, 610–615
 - defined, 535
 - fitting to 3D points, 729–731
 - linear component
 - intersection, 698–703
 - line intersection, 698–700
 - lozenge pseudodistance, 462
 - merging, 616–617
 - minimum-volume, 611–615
 - object culling by planes, 712
 - point in, 609–610
 - ray intersection, 700–701

- Spheres (*continued*)
 - segment intersection, 701–703
 - sphere pseudodistance, 461
 - triangle pseudodistance, 464
- Sphere-swept volumes, 538–539
 - defined, 538
 - line intersection, 703–709
 - medial set, 838
- Spherical dual
 - polyhedra, 396
 - spherical convex polygons, 401
 - tetrahedra, 397
- Spherical linear interpolation, 763
- Spherical polygons, 398, 399
- Split function, 357
- Splitting methods, 741
- Spot attenuation, 97–98
- Spot exponent, 98
- Spotlights
 - angle, 264, 265
 - defined, 93
 - plane and sphere
 - illumination, 908, 909
 - position, unit-length cone
 - axis, angle, 97
 - shader program, 904–905
 - See also* Lighting; Lights
- Sprites, 378
- Squared distance
 - line to line, 642–643
 - line to oriented box, 664–666
 - line to ray, 643–644
 - line to rectangle, 657–659
 - line to segment, 644
 - line to triangle, 651–654
 - minimum, 679
 - oriented box to oriented box, 670–672
 - point to circle, 675–676
 - point to line, 640
 - point to oriented box, 663
 - point to ray, 640–641
 - point to rectangle, 655–657
 - point to segment, 641
 - point to triangle, 646–651
 - ray to oriented box, 666
 - ray to ray, 645
 - ray to rectangle, 659–660
 - ray to segment, 645
 - ray to triangle, 654
 - rectangle to oriented box, 669–670
 - rectangle to rectangle, 661–663
 - rectangle to triangle, 661–663
 - segment to oriented box, 666–667
 - segment to rectangle, 660–661
 - segment to segment, 645–646
 - segment to triangle, 654–655
 - triangle to oriented box, 667–669
 - triangle to rectangle, 661–663
 - triangle to triangle, 661–663
- Square root, 754–755
- Stabbing, 481
- Standard coordinate system, 8, 9
- Static multitextures, 116
- Static objects, 392
- Static scene graphs, 307
- Static typecasting, 796
- Static visibility graph
 - defined, 489
 - pseudocode for computing, 490–491
 - vertices, 504, 505
 - See also* Visibility graphs
- Stationary objects, 390, 404–412
 - convex polygons, 404–409
 - convex polyhedra, 409–412
- Steady-state heat flow, 753
- Steepest descent search, 733–734
- Stencil buffers, 167–168
 - clearing, 179
 - comparison function, 167–168
 - defined, 167
 - functioning, 167–168
 - function use, 168
 - modification, 168
 - See also* Buffers
- Stencil test, 166
- Stream class
 - API, 809–812
 - defined, 809
 - interface, 809–810
 - object, 810
- Streaming, 808–819
 - defined, 809
 - functions, 812
 - memory, 811–812
 - to memory block, 816
 - Object API, 812–819
 - Stream API, 809–812
 - to/from disk, 810
 - usage, 810–811
 - See also* Object-oriented infrastructure
- Streaming SIMD Extensions (SSE) Driver, 143
- Stream macros, 799
- Sturm sequences, 739–740
- Subdivision
 - by arc length, 566–567
 - by midpoint distance, 567–568
 - by uniform sampling, 566
 - calculation in adjacent block, 601
 - for cubic curves, 568–570
 - curves, 566–570
 - fast, 568–570

- nonuniform, 594–596, 603–607
 - partial, 597, 598–599
 - rectangle patches, 587–602
 - recursive, 570
 - surfaces, 587–607
 - triangle patches, 602–607
 - Subpolygons, 402, 403
 - Subtree nodes, 275
 - Surface attributes, 92
 - Surface masses, 513–516
 - defined, 513
 - deformable, 514
 - illustrated, 513
 - See also* Mass-spring systems
 - SurfaceMesh class, 521
 - Surfaces, 573–607
 - Bézier rectangle patches, 574–578
 - Bézier triangle patches, 578–582
 - B-spline rectangle patches, 582–583
 - built from curves, 584–587
 - cylinder, 584–585
 - generalized cylinder, 585
 - implicit, 574
 - NURBS rectangle patches, 583–584
 - parametric patch, 573
 - parametric subdivision, 587–607
 - quadratic, 574
 - revolution, 586
 - tube, 586–587
 - SurRender Umbra, 376
 - Sweep
 - algorithm, 466
 - pseudocode, 467–468
 - SwitchNode class, 476
 - Symmetric matrix, 324
 - Symmetric triangulation, 607
 - System class, 515
 - Systems of equations, 719–722
 - linear systems, 719–720
 - polynomial systems, 720–722
 - See also* Numerical methods
 - System timer, 849
- T**
- Tagging, nodes, 311
 - Tangent line, 453, 454
 - Tangents, 756
 - defined, 917
 - incoming, 562
 - outgoing, 562
 - Tangent-space lighting, 916–919
 - coordinate system, 921
 - defined, 916
 - See also* Lighting
 - Target platforms, 310
 - Targets, 349
 - Templates, 800–801
 - for container classes, 801
 - defined, 800
 - library, 801
 - smart pointers, 803
 - use, 800
 - Temporal coherence, 469
 - Tension-continuity-bias splines, 562–566
 - TerminateFactory function, 812
 - Termination
 - application layer, 831
 - post-main, 825
 - system, 827
 - Terrain, 386–387
 - Test function, 457
 - TestIntersection function, 407–409, 410–411, 422, 427–428, 430–431, 441–443
 - Test-intersection queries, 390, 681, 686–693
 - line-capsule, 703
 - lines and OBBs, 686–688
 - line-sphere, 699
 - ray-capsule, 703–704
 - rays and OBBs, 688–691
 - ray-sphere, 700–701
 - segment-capsule, 704
 - segments and OBBs, 691–693
 - segment-sphere, 701–703
 - See also* Queries
 - Test method, 687
 - Tetrahedra
 - illustrated, 397
 - spherical dual, 397
 - vertices, 396
 - Text, 2D drawing and, 180
 - Texture coordinates
 - behavior, 106
 - clamped, 100, 102
 - clamp-to-edge mode, 103–104
 - color, 109
 - components, 100
 - computation, 109
 - defined, 99
 - mirror repeated, 101, 102
 - modes, 100–104
 - repeated, 101, 102
 - shader interpretation, 124
 - TextureEffect class, 298–299, 910
 - Texture filtering
 - anisotropic, 114, 115
 - bilinear, 106, 110
 - defined, 101
 - linear, 107
 - linear-linear, 111, 112
 - linear-nearest, 111
 - modes, 104–107
 - nearest-linear, 111
 - nearest-nearest, 110
 - nearest-neighbor, 105
 - trilinear, 112
 - Textures, 99–117
 - alpha values, 122

- Textures (*continued*)
 - clamped, 102
 - colors, 103
 - cube, 929
 - cylindrical, 101
 - depth, 99, 172, 946
 - effects, 304–305
 - mipmapping, 108–116
 - mirror repeated, 102
 - multitexture, 116–117, 911–913
 - pixel shader, 967
 - projected, 99, 943–945
 - realism, 99
 - repeated, 102
 - sampling, 107
 - shader programs, 909–910
- Texture samplers, 165, 208
- Texture sampling, 171–172
 - defined, 101, 171
 - implementation classes, 172
 - Wild Magic, 107, 171–172
- ThreeBuffers function, 160
- 3D objects, 7
- 3D picking, 247
- 3-tuples, 8, 17
- Top-down functional approach, 789
- Top-left rule, 163
- Top-level objects, 809, 815
- Top plane, 47
- Torsion, 543
- Trackball, 859, 865
 - motion, 861
 - projection, 860
 - rotation, 859–860
- Transformation class, 237–238, 359
 - Get functions, 239
 - member functions, 240
 - Set functions, 238–239
- Transformations, 7, 18–43, 234–244
 - affine, 29–31
 - applied during geometric pipeline, 61
 - homogeneous, 40–43
 - Householder, 722
 - inverse, 241–244
 - linear, 18–28
 - projective, 31–35
 - similarity, 861
 - Wild Magic support, 234
- Translation, 19, 30
 - backward, 856
 - forward, 856
 - in right direction, 852
 - speed, 856, 857
 - in up direction, 852
 - in view direction, 851
- Transporter class, 503, 504
- Transporters, 488
- Trapezoid rule, 744
- Tree manipulator, 346
- Tree structure, 230
- Triangle meshes, 99, 256–258
 - class, 257
 - defined, 256–257
 - drawing, 151
 - edge collapses, 381
 - edge-to-edge contact, 423
 - particle locations, 258
 - picking and, 478
 - shared pixels, 163
 - vertices, 258
 - See also* Meshes
- Triangle patches
 - Bézier, 578–582
 - defined, 574
- Triangle patch subdivision, 602–607
 - binary tree, 604
 - labeling, 604
 - nonuniform subdivision, 603–607
 - uniform subdivision, 602–603
 - See also* Subdivision
- Triangles, 533–534
 - back-facing, 69, 158
 - capsule pseudodistance, 464
 - clipped against bottom frustum plane, 75
 - clipped against left frustum plane, 76
 - clipped against right frustum plane, 76
 - clipped against top frustum plane, 75
 - clipped by one edge of rectangle, 157
 - clipping, 55
 - culling, 55
 - defined, 533
 - drawing as white object, 89
 - edge buffers, 160
 - front-facing, 69, 70
 - H-adjacent, 604, 605, 606
 - hypotenuse, 603
 - in/out of frustum, 70–71
 - linear components to, 651–655
 - lines to, 651–654
 - mass, 622, 623
 - model, 59
 - to oriented boxes, 667–668
 - points to, 646–651
 - project to triangles, 36, 38–39
 - pseudocode, 89–90
 - rasterization, 38, 89–92
 - rasterization with top-left rule, 163
 - rasterizer, 90–91
 - rays to, 654
 - to rectangles, 661–663
 - segments to, 654–655
 - sharing edges, 162
 - software-rendered image of, 62
 - sphere pseudodistance, 464
 - splitting configurations, 72

- to triangles, 661–663
 - vertices, 256
 - world, 59
 - Trilinear filtering, 112
 - Trilinear interpolation, 104
 - TriMesh class, 257
 - DoPick function, 476–477
 - objects, 309
 - Tube surfaces, 586–587
 - construction, 587
 - defined, 586–587
 - See also* Surfaces
 - Tuples, 8
 - Cartesian, 14
 - 3-tuples, 8, 17
 - 4-tuples, 17, 40, 129
 - TurnLeft function, 854
 - TurnRight function, 854
 - TwoBuffers function, 160
 - Two-point boundary value
 - problem, 747
 - Typecasting
 - dynamic, 793, 796
 - static, 796
- U**
- Underscore, 792
 - Undershooting, 563
 - Uniform knots, 322
 - Uniform scaling
 - commutativity, 237
 - defined, 25
 - inversion, 235
 - See also* Scaling
 - Uniform subdivision, 566, 567
 - rectangle patches, 587–594
 - triangle patch subdivision, 602–603
 - See also* Subdivision
 - Unique identifiers, 820–821
 - Unit-length eigenvectors, 619
 - UpdateBS function, 271, 278
 - Update function, 350, 527
 - UpdateGS function, 271, 274, 277, 278, 280
 - UpdateModeNormals function, 272
 - UpdateMS function, 272
 - UpdatePointMotion function, 350
 - UpdateRS function, 281, 284, 286, 303
 - Updates
 - controller, 277
 - dynamic, 317, 521
 - geometric-state, 268–280
 - occurrence with geometric
 - quantity change, 279
 - render-state, 280–289
 - scene graph, 268–289
 - as side effects to changes, 279
 - world bounding volume, 277
 - UpdateState function, 282
 - UpdateSystemMotion function, 350
 - UpdateWorldBound function, 272
 - UpdateWorldData function, 272, 275, 359
 - Up direction, 8
 - UserConstant class, 207–208, 209, 211
 - User-defined constants, 192, 207–208
 - objects, 209
 - shader program association, 209
 - User-defined maps, 375
- V**
- Validation, of shader programs, 213–216
 - Vector class, 16–17
 - Vector2 class, 130
 - Vector4 class, 132
 - Vectors
 - direction, 857
 - fixed up, 571
 - inverse transformation, 241
 - knot, 332, 552–553, 558–559
 - matrix application, 129, 130
 - on-the-right notation, 130
 - perpendicular direction, 534
 - points and, 15–17
 - reflection, 928
 - right, 857
 - rotation, 759–782
 - squared length, 725
 - squares, summing, 677
 - storage in linear memory, 132
 - up, 857
 - Vector-valued equations, 323
 - Verifiability, software, 784
 - Vertex attributes, 92–125
 - colors, 92
 - computation, 156
 - defined, 92
 - fog, 122–123
 - interpolation, 123
 - lighting and materials, 92–99
 - multipass effects, 188
 - rasterizing, 124–125
 - storage, 91
 - textures, 99–117
 - transparency, opacity,
 - blending, 117–122
 - Vertex buffers, 222
 - functions associated with, 185
 - internal, 189
 - loading, 188
 - See also* Buffers
 - Vertex.cg, 897
 - VertexColor3Effect class, 899
 - VertexColor4Effect class, 899
 - Vertex colors, 897–899
 - RGB, 898
 - RGBA, 898
 - Vertex controllers, 225
 - Vertex morphing, 316, 349–350

- VertexProgram class, 205
 - Vertex program loader, 185
 - Vertex shader programs, 68, 97, 249–251, 299
 - applying, 149
 - enabler, 192–193
 - for lighting, 303
 - outputs, 149–150
 - pass-through, 150
 - in shallow wrapper, 151
 - Vertex (vertices)
 - clipped convex polygon, 156
 - clustering, 380
 - color, 92, 161
 - convex, 499, 500
 - current, 500–501
 - decimation, 380
 - indices, 158
 - minimum-estimate, 493
 - outer edges, 501
 - of parabola, 323
 - recording, 385–386
 - skin, 348
 - static visibility graph, 504, 505
 - tetrahedra, 396
 - visibility graph, 489, 490, 493
 - Video RAM (VRAM), 174, 183
 - image loading to, 254
 - limited amount of, 353
 - View coordinates, 51
 - View direction, 8
 - View frustum
 - clipping to, 70–77
 - defined, 43
 - eye point and, 44
 - full viewport of, 48
 - 3D drawing, 46
 - View matrices, 51–52, 134–136
 - View plane, 43
 - Viewports
 - defined, 43
 - entire, for drawing, 473
 - normalized coordinates, 473
 - on near plane, 474
 - settings, 473
 - of view frustum, 48
 - View space, 8, 50–52
 - defined, 50
 - normal, 927
 - View volume, 43
 - Visibility graphs, 482, 489–494
 - adjacency matrix, 490
 - construction pseudocode, 505–506
 - Dijkstra’s algorithm, 492–494
 - dynamic, 491–492
 - efficient calculation, 504–506
 - static, 489
 - vertex values, 493
 - vertices, 489, 490
 - VisibleObject class, 289
 - VisibleSet class, 289, 290
 - Volume masses, 516–519
 - defined, 516
 - deformable, 517
 - See also* Mass-spring systems
 - Volumetric fog, 947–950
 - defined, 947
 - fog factor, 949
 - fog generator, 947–948
 - height field rendered with, 949
 - shader programs, 948–950
- W–X**
- Walls, 483
 - Water effects, 955–956
 - pixel shader, 955
 - rippling ocean application, 956
 - vertex shader, 955
 - Wave and shock phenomena, 752
 - Waypoints, 486
 - Wedge class, 505
 - WglRenderer class, 176, 180
 - WhichSide function, 250, 292, 363
 - Which-side-of-plane query, 245, 250
 - Wild Magic
 - application layer, 65, 831
 - application types, 832
 - colors, 183
 - as commercial-quality engine, 128
 - controllers support, 316
 - conventions, 128–145
 - cube-map sampler, 929
 - culling, 289
 - default bounding volume type, 247
 - effect rendering, 304
 - engine, 128
 - fast computations, 144, 145
 - geometric primitives, 195
 - isotropic mipmapping implementation, 116
 - lighting vertex shaders, 97, 98
 - lights implementation, 94
 - matrix composition, 134
 - memory manager, 876
 - nonuniform scaling and, 234, 236
 - portal system, 367
 - rendering layer, 174
 - resource management system, 174
 - rotations, 141
 - scene graph files, 309
 - scene graph management, 220
 - shader creation in, 957–971
 - software renderer, 61
 - texture sampling, 107, 171–172
 - transformations, 234
 - vector/matrix conventions, 8

- vector-on-right, row-major order, 133
 - versions, 3
 - view matrices, 135
 - window handedness, 140
 - `WindowApplication3` class, 849–866
 - camera motion, 851–857
 - defined, 849
 - interface, 849–851
 - object motion, 857–865
 - performance measurements, 865–866
 - `WindowApplication` class, 842–849
 - constructor, 844
 - event callbacks, 845
 - event handling, 845–849
 - interface, 842–883
 - member set access, 844–845
 - object, 845
 - Window coordinates
 - defined, 57
 - floating-point, 163
 - Windowed applications, 842–849
 - Windows
 - handedness, 139–140
 - matrix, 57
 - width, 153
 - Window space, 8, 56–58
 - clip space to, 164
 - defined, 57
 - Wireframe mode
 - `DrawTriMesh` function, 157–158
 - `RasterEdge` function, 164
 - toggling to, 164
 - World bound, 223
 - World coordinates
 - generation, 860
 - inertia tensor in, 523
 - rotation axis, 860
 - system, 482
 - World matrix, 50
 - World space, 8, 48–50
 - defined, 48
 - object in, 49
 - problem, 48
 - World transformations, 223
 - controller in computation, 276
 - for DAG, 231
 - defined, 49
 - directly setting, 270
 - memory usage, 236
 - `Spatial` class, 270
 - Worst fit policy, 890
- Y–Z**
- Yaw, 858
 - Z-buffer state, 282, 284
 - Z-test, 168