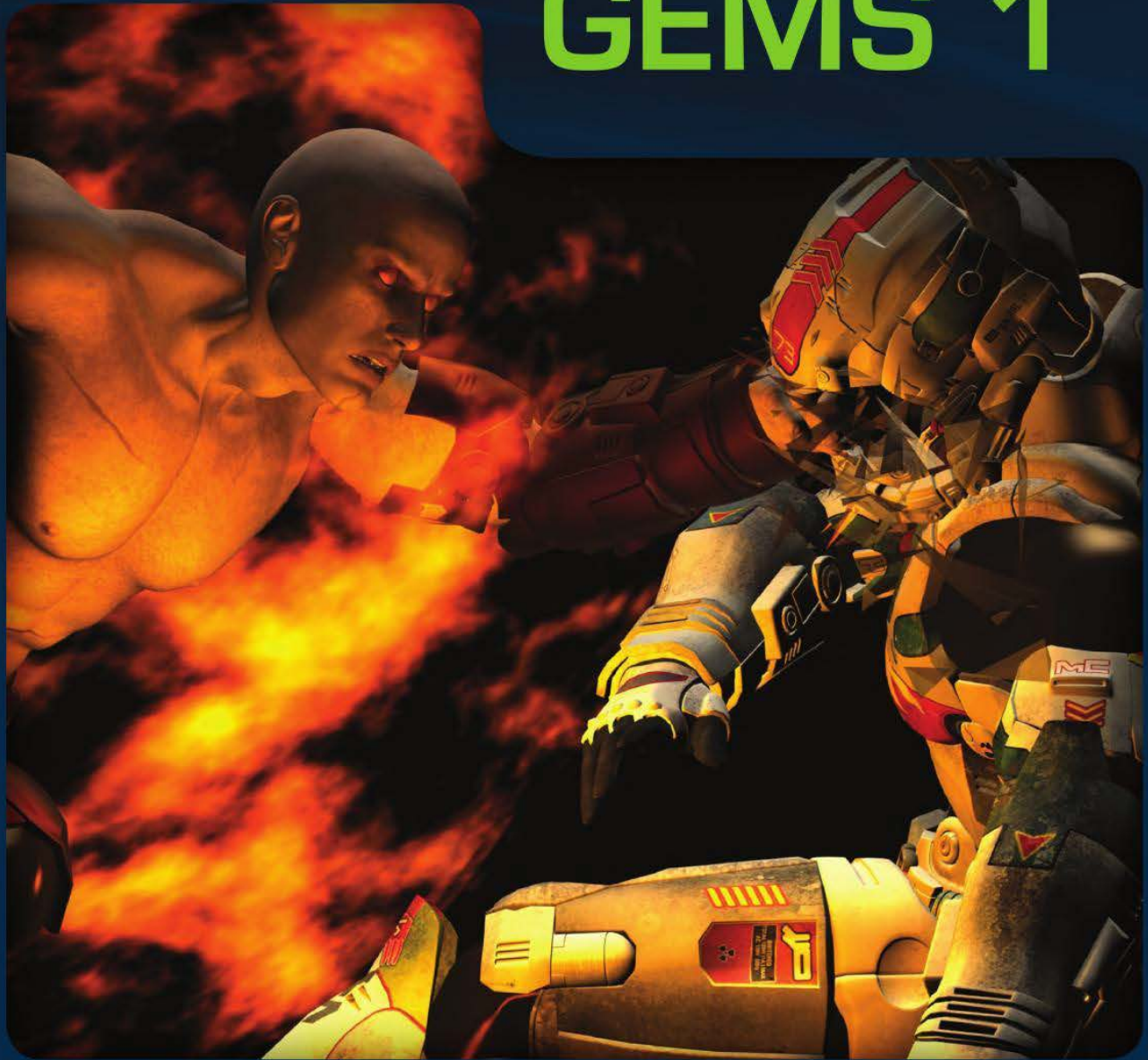


GAME ENGINE

GEMS 1



SERIES EDITOR: ERIC LENGYEL

Game Engine Gems, Volume One

Edited by Eric Lengyel,

Ph.D.



JONES AND BARTLETT PUBLISHERS

Sudbury, Massachusetts

BOSTON, TORONTO, LONDON, SINGAPORE

World Headquarters

Jones and Bartlett Publishers
40 Tall Pine Drive
Sudbury, MA 01776
978-443-5000
info@jbpub.com
www.jbpub.com

Jones and Bartlett Publishers Canada
6339 Ormindale Way
Mississauga, Ontario L5V 1J2
Canada

Jones and Bartlett Publishers International
Barb House, Barb Mews
London W6 7PA
United Kingdom

Jones and Bartlett's books and products are available through most bookstores and online booksellers. To contact Jones and Bartlett Publishers directly, call 800-832-0034, fax 978-443-8000, or visit our website, www.jbpub.com.

Substantial discounts on bulk quantities of Jones and Bartlett's publications are available to corporations, professional associations, and other qualified organizations. For details and specific discount information, contact the special sales department at Jones and Bartlett via the above contact information or send an email to specialsales@jbpub.com.

Copyright © 2011 by Jones and Bartlett Publishers, LLC

ISBN-13: 9780763778880
ISBN-10: 0763778885

All rights reserved. No part of the material protected by this copyright may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products. All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks, etc., is not an attempt to infringe on the copyright of others.

Production Credits

Publisher: David Pallai

Editorial Assistant: Molly Whitman

Senior Production Editor: Katherine Crighton

Associate Marketing Manager: Lindsay Ruggiero

V.P., Manufacturing and Inventory Control: Therese Connell

Cover Design: Kristin E. Parker

Cover Image: © Jesse-lee Lang/Dreamstime.com

Printing and Binding: Malloy, Inc.

Cover Printing: Malloy, Inc.

6048

Printed in the United States of America

14 13 12 11 10 10 9 8 7 6 5 4 3 2 1

Contents

Introduction	x
Contributor Biographies.....	xii
About the Editor	xx

Part I Game Engine Design

Chapter 1 What to Look for When Evaluating Middleware for Integration

1.1 Middleware, How Do I Love Thee?	3
1.2 Integration Complexity and Modularity	4
1.3 Memory Management	4
1.4 Mass Storage I/O Access.....	6
1.5 Logging.....	7
1.6 Error Handling.....	7
1.7 Stability and Performance Consistency.....	8
1.8 Custom Profiling Tools.....	9
1.9 Customer Support	9
1.10 Demands on the Maintainers	10
1.11 Source Code Availability	10
1.12 Quality of Source Code.....	11
1.13 Platform Portability.....	12
1.14 Licensing Requirements	12
1.15 Cost.....	13

Chapter 2 The Game Asset Pipeline.....

2.1 Asset Pipeline Overview	16
2.2 Asset Pipeline Design.....	24
2.3 Push or Pull Pipeline Model	28
2.4 COLLADA, A Standard Intermediate Language.....	31
2.5 OpenCOLLADA.....	42
2.6 User Content.....	46
Chapter 3 Volumetric Representation of Virtual Environments	53
3.1 Introduction.....	53
3.2 Overview.....	55
3.3 Data Structures	58
3.4 Surface Extraction	63
3.5 Rendering	71
3.6 Physics.....	78
3.7 The Future.....	79
Chapter 4 High-Level Pathfinding	83
4.1 Terms	84
4.2 Start Your Engines	85
4.3 Why High-Level Pathfinding?.....	86
4.4 Preprocess Phase	87
4.5 Fuzzy Pathing Phase	97
4.6 Detailed Paths Phase.....	102
4.7 Why Go Through All This Trouble?	104
Chapter 5 Environment Sound Culling.....	107
5.1 The Problem.....	108
5.2 A Sound Culling Solution	110

5.3 Constructing the Sound Grid.....	112
5.4 Processing the Sound Grid.....	114
5.5 Supporting Multiple Listeners	121
5.6 Extensions.....	121
Chapter 6 A GUI Framework and Presentation Layer	123
6.1 GUI Systems.....	123
6.2 Design Patterns: Model View Controller (MVC)	125
6.3 A GUI Design.....	127
6.4 And Finally	140
Chapter 7 World's Best Palettizer	143
7.1 Palettes? Whatever for?.....	143
7.2 Understanding Quantization	146
7.3 Hard-Earned Lessons.....	147
7.4 Algorithm Overview.....	149
7.5 Future Work.....	160
7.6 Results	161
Chapter 8 3D Stereoscopic Rendering: An Overview of Implementation Issues	165
8.1 Mechanisms of Plano-Stereoscopic Viewing.....	166
8.2 Stereo Techniques.....	176
8.3 Design Considerations for 3D Scenes.....	179
8.4 Outlook	182
Chapter 9 A Multithreaded 3D Renderer	185
9.1 The Memory Model	186
9.2 Building the Display Lists in Parallel	188

9.3 Parallel Models.....	190
9.4 Synchronizing the GPU and CPU.....	191
9.5 Using Additional Processing Resources.....	192
9.6 Reducing the Pressure on the Memory Bandwidth.....	193
9.7 Performing Graphical Operations in Parallel.....	194
Chapter 10 Camera-Centric Engine Design for Multithreaded Rendering.....	197
10.1 Uses of Multi-Core in Video Games	198
10.2 Multithreaded Command Buffers.....	200
10.3 Device-Independent Command Buffers	201
10.4 A Camera-Centric Design.....	207
10.5 Future Work.....	217
Chapter 11 A GPU-Managed Memory Pool.....	219
11.1 Background	220
11.2 The Memory Pool	221
11.3 Synchronization Issues	223
11.4 The Staging Buffer.....	225
11.5 Memory Pool Defragmentation	227
11.6 Memory Pool Eviction.....	228
11.7 Platform-Specific Considerations	229
11.8 Future Work.....	230
Chapter 12 Precomputed 3D Velocity Field for Simulating Fluid Dynamics	233
12.1 Introduction.....	233
12.2 Velocity Field Computation	235
12.3 Physics Simplification.....	239
12.4 Results and Discussion	242

Chapter 13 Mesh Partitioning for Fun and Profit	245
13.1 Desirable Algorithm Properties.....	246
13.2 Lessons Learned.....	248
13.3 When Greedy Is Good	250
13.4 Future Work.....	252
13.5 Graphical Walkthrough.....	253
 Chapter 14 Moments of Inertia for Common Shapes	 259
14.1 Center of Mass	259
14.2 The Inertia Tensor.....	261
14.3 Derivation of Moments of Inertia	264
14.4 Summary.....	284
 Part II Rendering Techniques	 287
<hr/>	
 Chapter 15 Physically-Based Outdoor Scene Lighting	 289
15.1 Positioning the Sun and Moon	290
15.2 Computing Natural Sunlight	291
15.3 Moonlight and Other Nighttime Light Sources	295
15.4 Tone-Mapping the Light	296
15.5 Implementation Notes	299
 Chapter 16 Rendering Physically-Based Skyboxes	 301
16.1 Generating and Drawing the Skybox.....	301
16.2 Computing the Skybox Vertex Colors.....	303
16.3 Integrating the Skybox with Your Scene	307
16.4 Embellishing Your Skybox	308

Chapter 17 Motion Blur and the Velocity-Depth-Gradient Buffer	311
17.1 Technique Overview	312
17.2 Rendering to the Velocity-Depth-Gradient Buffer.....	315
17.3 Rendering the Post-Processing Effect	321
17.4 Grid Optimization.....	325
Chapter 18 Fast Screen-Space Ambient Occlusion and Indirect Lighting	327
18.1 Introduction.....	327
18.2 A General Ambient Illumination Model	329
18.3 Screen-Space Representation of the Scene.....	332
18.4 Volumetric Ambient Occlusion	333
18.5 Indirect Lighting of the Near Geometry	338
18.6 Implementation	338
18.7 Results.....	341
Chapter 19 Real-Time Character Dismemberment	345
19.1 What is Character Damage Modeling?	346
19.2 Methods of Mutilation.....	347
19.3 Bone Matrix Flattening.....	348
19.4 Improvements.....	350
19.5 Demo.....	353
Chapter 20 A Deferred Decal Rendering Technique.....	355
20.1 The Problem.....	356
20.2 The General Idea	357
20.3 Geometry Rendering.....	359
20.4 Fade Out And Wrap-Around.....	361
20.5 Surface Clipping	364

20.6 Limitations	365
20.7 Additional Features	367
Part III Programming Methods.....	369
<hr/>	
Chapter 21 Multithreaded Object Models	371
21.1 Explicit Locking	372
21.2 Message-Based Updates.....	373
21.3 Multiple Thread Contexts	374
21.4 Buffered State Changes	375
21.5 Selecting the Best Approach.....	378
Chapter 22 Holistic Task Parallelism for Common Game Architecture Patterns.....	381
22.1 Tasks Versus Threads in Games.....	381
22.2 The Task Scheduler	383
22.3 Decomposing Game Patterns into Tasks	384
22.4 The Future of Task Parallelism in Games	390
Chapter 23 Dynamic Code Execution Hierarchies	391
23.1 What are Code Execution Hierarchies?	392
23.2 Design Features.....	395
23.3 Benefits & Pitfalls.....	399
Chapter 24 Key-Value Dictionary.....	401
24.1 Design	401
24.2 Using the KVD.....	402
24.3 Code Details	404
24.4 Caveats	407

Chapter 25 A Basic Scheduler	409
25.1 Overview.....	409
25.2 Task Functionality.....	410
25.3 Scheduler Functionality.....	411
25.4 Implementation	413
25.5 Additional Functionality	413
 Chapter 26 The Game State Observer Pattern	 415
26.1 Creating a Game State Manager.....	418
26.2 The Interfaces of the Game State Observer Pattern	422
26.3 Making GameState Observable	424
26.4 Creating Observers.....	427
26.5 Managing Functionality by Game State	430
 Chapter 27 Fast Trigonometric Operations Using CORDIC Methods	 433
27.1 Rotation Mode Algorithm	434
27.2 Vectoring Mode Algorithm	436
27.3 Applications	437
27.4 Implementation	437
27.5 Considerations.....	443
27.6 Extensions	443
 Chapter 28 Inter-Process Communication Based on Your Own RPC Subsystem	 445
28.1 History of Remote Procedure Call.....	447
28.2 How RPC Works: Internal Architecture of RPC	447
28.3 How to Build Your Own RPC Subsystem.....	451
28.4 Why RPC is Useful for Game Engines	455

Introduction

In the fields of computer graphics and computer game development, the word *gem* has been established as a term for describing a short article that focuses on a particular technique, a clever trick, or practical advice that a person working in these fields would find interesting and useful. The term *gem* was first used in 1990 for the first volume of the *Graphics Gems* series of books, which concentrated on knowledge pertaining to computer graphics. The mainstream methods for rendering 3D images have changed considerably since then, but many of those gems still comprise useful techniques today and have demonstrated a timeless quality to the knowledge they contain. Several newer book series containing the word "Gems" in their titles have appeared in related subject areas such as game programming and GPU rendering, and they all advance the notion of sharing knowledge through concise articles that each focus on a specific topic. We continue the tradition with this book, the first volume of *Game Engine Gems*.

Game Engine Gems concentrates on knowledge relating to the development of game engines, which encompass the architecture, design, and coding methods constituting the technological foundation for today's video games. A complete game engine typically includes large components that handle graphics, audio, networking, and physics. There may also be large components that provide services for artificial intelligence (AI) and graphical user interfaces (GUIs), as well as a variety of smaller components that deal with resource management, input devices, mathematics, multithreading, and many additional pieces of generic functionality required by the games built upon them. Furthermore, many game engines are able to run on multiple platforms, which may include PCs and one or more game consoles such as the PlayStation 3 or Xbox 360. The *Game Engine Gems* series is specifically intended to

include all such aspects of game engine development targeting all current game platforms.

This book is divided into three parts covering the broad subject areas of game engine design, rendering techniques, and programming methods. The 28 gems appearing in this book are written by a group of 25 authors having expertise in game engine development, some quite extensive. It is our hope that the wisdom recorded in these pages and the pages of future volumes of *Game Engine Gems* continue to serve game developers for many years to come.

Call for Papers

At the time this book is published, work on the second volume of *Game Engine Gems* will have already entered its early stages. If you are a professional developer working in a field related to game development and would like to submit a contribution to the next book in the series, please visit our official website at <http://www.gameenginegems.com/>.

Contributor Biographies

Rémi Arnaud remi@acm.org

Rémi Arnaud is working as Chief Software Architect at Screampoint International, a company providing interoperable 5D digital city models for the benefit of governments, property owners, developers, designers, contractors, managers, and service providers. Rémi's involvement with real-time graphics started in the R&D department of Thomson Training & Simulation (now Thales) designing and then leading the Space Magic real-time visual system for training simulators, where he finalized his Ph.D. "La synthèse d'images en temps réel". He then relocated to California to join the Silicon Graphics IRIS Performer team, working on advanced features such as calligraphic light points for training pilots. He then decided to be more adventurous and co-founded Intrinsic Graphics, where he co-designed the Alchemy engine, a middleware targeting cross-platform game development for PS2, Xbox, GameCube, and PC. He was hired as Graphics Architect at Sony Computer Entertainment US R&D, working on the PlayStation 3 SDK graphics API, and joined the Khronos Group to create COLLADA asset exchange standard. More recently, Rémi worked at Intel where he created and lead the Larrabee Game Engine Technology team.

Ron Barbosa ron@exibeo.net

Ron Barbosa has been an avid hobbyist game and game technology developer since his teenage years. Since 1993, he has worked as a professional network/software engineer for many companies producing internet technologies, including former technology giants Compaq Computer Corporation and Lucent Technologies, Inc. He

currently serves as the Chief Software Architect at Boca Raton, Florida's Revelex Corporation, a travel technology services provider. In his short spurts of spare time, he attempts to remain active in indie game development circles and is the original author of *Planet Crashmania 9,000,000* available on Microsoft's Xbox LIVE Indie Games service and Apple's iPod Touch Apps Store (ported to iPod Touch by James Webb).

John Bolton johnjbolton@yahoo.com

John Bolton is a software engineer at Netflix in Los Gatos, California and has been programming games professionally since 1992. He has contributed to dozens of games and has been lead programmer on several titles, including *I Have No Mouth and I Must Scream*, *Heroes of Might and Magic*, and *High Heat Baseball*.

Khalid Djado Khalid.Djado@USherbrooke.ca

Khalid Djado is a Ph.D. student in the Department of Computer Sciences at University of Sherbrooke. His research interests include computer graphics and physical simulations. He is a lecturer for graduate students in game development for the University of Sherbrooke at Ubisoft Campus. He was also a game developer at Amusement Cyanide in Montreal. He obtained a bachelor's degree in applied mathematics from the University Sidi Mohamed Ben Abdellah in Morocco, and a master's in modelling, simulation, and optimisation from the University of Bretagne Sud in France. He has been a member of ACM Siggraph since 2006.

Richard Egli Richard.Egli@USherbrooke.ca

Richard Egli is professor in the Department of Computer Sciences at University of Sherbrooke since 2000. He received his B.Sc. degree in Computer Science and his M.Sc. degree in Computer Sciences at University of Sherbrooke (Québec, Canada). He received his Ph.D. in Computer Sciences from University of Montréal (Québec, Canada) in 2000. He is the director of the centre MOIVRE (MOdélisation en Imagerie, Vision et

RÉseaux de neurones). His research interests include computer graphics, physical simulations, and digital image processing.

Simon Franco simon_franco@hotmail.com

Simon Franco sampled his first taste of programming on the Commodore Amiga, when he wrote his first *Pong* clone in AMOS, and he has been coding ever since. He joined the games industry in 2000 after completing a degree in computer science. He started at The Creative Assembly in 2004, where he has been to this day. When he's not playing the latest game, he'll be writing assembly code for the ZX spectrum.

Anders Hast aht@cb.uu.se

Anders Hast works half time as a Visualization Expert at UPPMAX (Uppsala Multidisciplinary Center for Advanced Computational Science) and half time as associate professor at the University of Gävle, both in Sweden. He has published well over 50 scientific papers in journals, in conferences, and as book chapters in various areas in computer graphics, visualization, and applied mathematics. His other interests in life, besides computer graphics research, are US model trains, drinking Czech beer, and studying the Italian language.

Daniel F. Higgins webmaster@programming.org

Dan has spent over 10 years in the games industry, starting with Stainless Steel Studios. He was one of the original creators of the Titan game engine, and was one of the chief AI programmers on *Empire Earth*, *Empires: Dawn of the Modern World* and *Rise & Fall: Civilizations at War*. Later, he worked at Tilted Mill on *Caesar IV* and *SimCity Societies*. Today, along with his wife, he is owner and manager of Lunchtime Studios, Inc.

Adrian Hirst adrian@weaseltron.com

Adrian Hirst has been shedding blood, sweat, and tears programming on any and every gaming platform for the last ten years, working with many leading developers and publishers, most recently including Sony, Codemasters (*LMA 2002*, *Colin McRae PC 3*, 4, 5, 2005+), and Electronic Arts/Criterion (*Burnout: Paradise*). Most recently he set up Weaseltron Entertainment in order to join the growing masses of independent developers and apply his skills to new challenges. He is also remarkably good looking, writes his own biography, and needs a beer.

Jason Hughes jhughes@steelpennygames.com

Jason Hughes is an industry veteran game programmer of 15 years and has been actively coding for 25 years. His background covers everything from modem drivers in 6502 assembly to fluid dynamics on the Wii to a multi-platform 3D engine. Jason tinkers with exotic data structures, advanced compression algorithms, and various tools and technology relating to the games industry. Prior to founding Steel Penny Games, Jason spent several years at Naughty Dog on the ICE team writing the asset pipeline tools used by PS3 developers in the ICE and Edge libraries.

Frank Kane fkane@sundog-soft.com

Frank Kane is the owner of Sundog Software, LLC, makers of the *SilverLining* SDK for real-time rendering of skies, clouds, and precipitation effects (see www.sundog-soft.com for more information). Frank's game development experience began at Sierra On-Line, where he worked on the system-level software of a dozen classic adventure game titles including *Phantasmagoria*, *Gabriel Knight II*, *Police Quest: SWAT*, and *Quest for Glory V*. He's also an alumnus of Looking Glass Studios, where he helped develop *Flight Unlimited III*. Frank developed the C2Engine scene rendering engine for SDS International's Advanced Technology Division, which is used for virtual reality training simulators by every branch of the US military. He currently lives with his family outside Seattle.

Jan Krassnigg Jan@Krassnigg.de

Jan Krassnigg is studying Information Technologies at the University of Aachen, Germany.

Martin Linklater mslinklater@mac.com

Martin Linklater has been programming since 1981 when he was ten years old. After spending his teenage years hacking C64 and Amiga code, he got a Bachelors Degree in Computer Science in 1993. His first job in the games industry was as a programmer for Psygnosis, soon to become Sony Computer Entertainment Europe. After five years at SCEE he left with five colleagues to start Curly Monsters, an independent development house. Curly Monsters closed in 2003 after releasing two titles. Martin worked for a short time for EA, then returned to Sony in 2003. Martin is currently a Technical Director working on an undisclosed Sony title. Martin lives in Wallasey, UK with his wife and two-year-old son. He enjoys games, flight simulation, sim racing, and beer.

Colt McAnlis duhroach@gmail.com

Colt McAnlis is a graphics programmer at Blizzard Entertainment, where he works on stuff he typically can't talk about. Prior, Colt was a graphics programmer at Microsoft Ensemble studios, where in his free time he moonlighted as an Adjunct Professor at SMU's GUILDHALL school for video game development. He has received a number of publications in various industry books, and continues to be an active participant in speaking at conferences.

Jeremy Moore jeremy.moore@disney.com

Jeremy Moore is the lead engine programmer for the Core Technology Group at Disney's Black Rock Studio in Brighton, UK. He has been working in the games industry for over a decade. Four of those years were spent working on SCEA's *ATV Offroad Fury*

games on both PS2 and PSP. Among other things, he was responsible for the acclaimed network play implementation. He now specializes in real-time graphics and being ordered around by his two young daughters.

Jon Parise jon@indelible.org

Jon Parise is a senior software engineer at Electronic Arts. He has worked on a number of titles, including *The Sims 3*, *The Lord of the Rings: The White Council*, *Ultima Online*, and *The Sims Online*. He was also a contributing author for *Massively Multiplayer Game Development 2*. Jon earned a bachelors degree in Information Technology from the Rochester Institute of Technology and a masters degree in Entertainment Technology from Carnegie Mellon University.

Kurt Pelzer kurt.pelzer@gmx.net

Kurt Pelzer is a Senior Software Engineer and Software Architect with a decade of experience in team-oriented projects within the 3D real-time simulation and games industry. At Piranha Bytes, he has taken part in the development of the games *Risen* (PC & Xbox 360), *Gothic 1-3* (PC) and the engine technology used for these products. Kurt has published articles in the technical book series *GPU Gems*, *Game Programming Gems*, and *ShaderX*.

Aurelio Reis AurelioReis@gmail.com

Aurelio Reis is a programmer at id Software, where he works on graphics and special effects. While he's interested in all aspects of game development, he especially enjoys working on networking and gameplay as well as doing research on cutting edge graphics techniques. An industry veteran and avid gamer, Aurelio has contributed to numerous titles over the years, but is most excited about the game he's working on right now, *Doom 4*.

Sébastien Schertenleib sscherten@bluewin.ch

Sébastien Schertenleib has been involved in academic research projects creating 3D mixed reality systems using stereoscopic visualization while completing his Ph.D. in Computer Graphics at the Swiss Institute of Technology in Lausanne. Since then, he has been holding a job as a Principal Engineer at Sony Computer Entertainment Europe's R&D Division. This role includes supporting game developers on all PlayStation platforms by providing technical training, presenting at various games conferences, and working directly with game developers via on-site technical visits and code share.

László Szirmay-Kalos szirmay@iit.bme.hu

László Szirmay-Kalos is the head of the Department of Control Engineering and Information Technology at the Budapest University of Technology and Economics. He received his Ph.D. in 1992 and full professorship in 2001 in computer graphics. His research area is Monte-Carlo global illumination algorithms and their GPU implementation. He has more than two hundred publications in this field. He is the fellow of *Eurographics*.

Balázs Tóth tbalazs@sch.bme.hu

Balázs Tóth is an assistant processor at the Budapest University of Technology and Economics. He is involved in distributed GPGPU projects and deferred shading rendering and is responsible for the CUDA education of the faculty.

Tamás Umenhoffer umitomi@gmail.com

Tamás Umenhoffer is an assistant processor at the Budapest University of Technology and Economics. His research topic is the computation of global illumination effects and realistic lighting in participation media and their application in real-time systems and games.

Brad Werth bradley.j.werth@intel.com

Brad Werth is a Senior Software Engineer in Intel's Visual Computing Division. He has been a frequent speaker at the Game Developers Conference and Austin GDC.

David Williams david@david-williams.info

David Williams received his M.Sc. in Computer Science from the University of Warwick in 2004 before joining City University as a Ph.D. student researching Medical Visualization. It was at this point that he developed an interest in voxels and began investigating how the concepts could be applied to game engines. He received his Ph.D. in 2008, but has continued to work on his Thermite3D voxel engine in his spare time. He now works as a graphics programmer for a game development company in the UK, and also enjoys photography and travelling.

About the Editor

Eric Lengyel lengyel@terathon.com

Eric Lengyel is a veteran of the computer games industry with over 15 years of experience writing game engines. He has a Ph.D. in Computer Science from the University of California, Davis, and he has a Masters Degree in Mathematics from Virginia Tech. Eric is the founder of Terathon Software, where he currently leads ongoing development of the C4 Engine.

Eric entered the games industry at the Yosemite Entertainment division of Sierra Online in Oakhurst, California, where he was the lead programmer for the fifth installment of the popular adventure RPG series *Quest for Glory*. He then worked on the OpenGL team for Apple Computer at their headquarters in Cupertino, California. More recently, Eric worked in the Advanced Technology Group at Naughty Dog in Santa Monica, California, where he designed graphics driver software used on the PlayStation 3 game console.

Eric is the author of the bestselling book *Mathematics for 3D Game Programming and Computer Graphics*. He is also the author of *The OpenGL Extensions Guide*, the mathematical concepts chapter in the book *Introduction to Game Development*, and several articles in the *Game Programming Gems* series. His articles have also been published in the *Journal of Game Development*, in the *Journal of Graphics Tools*, and on *Gamasutra.com*. Eric currently serves on the editorial board for the recently renamed *Journal of Graphics, GPU, and Game Tools* (JGGGT).

About the CD

The accompanying CD contains supplementary materials for many of the gems in this book. These materials are organized into folders having the chapter numbers as their names. The contents include demos, source code, examples, specifications, and larger versions of many figures. For chapters that include project files, the source code can be compiled using Microsoft Visual Studio.

High-resolution color images are included on the CD for many chapters, and they can be found in the folders named Figures inside the chapter folders. All of the figures shown in the color plates section of this book are included. Additionally, color versions of figures are included from several additional chapters that were only printed in black and white.

Part I

Game Engine Design

1

What to Look for When Evaluating Middleware for Integration

Jason Hughes

Steel Penny Games, Inc.

1.1 Middleware, How Do I Love Thee?

Modern games are very rarely works comprised entirely of proprietary, custom code written by in-house developers. The amount of polished functionality required to compete in the games industry is simply an enormous task for a single studio to undertake, and is in a word, unproductive. These days, game programmers are expected to be comfortable using certain wheels that have been invented elsewhere and only reinventing those that provide tangible benefits to their project or studio by directly contributing to the success of a title or differentiating them in some way from the competition. Given that some middleware libraries will be chosen to fulfill certain needs, and often there are several products to choose between with similar feature sets, we ask, "What are the considerations a team should take into account when comparing middleware products?"

Let's assume that the language of choice is C/C++, as is most common today with middleware and game development. Let's also assume that we're discussing an existing

codebase where a specific feature is missing that can be filled by middleware, and that the team's desire is to be surgical during integration and leave the smallest scar possible should it need to be removed.

1.2 Integration Complexity and Modularity

The first, most important feature of a middleware package is its integration complexity. Good libraries are highly modular, minimally intrusive, and easy to plug into very different codebases. In short, they make very few assumptions and are fairly decoupled from implementation details of other systems. A good library should come with reasonable defaults or require very limited configuration before the system is in a testable, working state. The integration engineer's level of experience can play a role in the usability of a system, so minimal integration is key. This promotes a rapid evaluation cycle—it leaves a bitter aftertaste when an engineer has a long integration cycle, only to find the library is not desirable. My rule of thumb is two days. Anything that takes more than two days to get running will waste a week putting it through its paces, and there aren't enough weeks in a development cycle to try out alternatives.

1.3 Memory Management

Console developers all know that careful memory management is crucial to a stable product. Middleware intended for virtual memory-backed PC systems, or even those technologies with hefty demands that next generation consoles can satisfy, may not be suitable for the more modest RAM budgets of yesteryear. It's important to know not only what memory budgets are expected, but also who is responsible for managing the allocations.

Ideally, each middleware library will have its own memory management scheme

that simply initializes with two arguments: a pointer to a block of memory and its size. A middleware library author is the person most knowledgeable of the sizes of allocations, the churn rate for allocations, and the memory management algorithm most appropriate to prevent fragmentation within their own allocation pool. Further, this has the advantage of explicitly notifying the developer when the memory budget has been exceeded, rather than collecting undesirably large amounts of memory from the main heap where tracking down memory consumption can be tedious and problematic. Also, this method tends to highlight integration mishaps where library assets are being leaked, because it soon fails when the heap is exhausted. A resizable heap is sometimes desirable, specifically if certain game levels need to shift memory privileges to emphasize different systems, though this feature is fairly rare.

In the absence of a completely isolated heap managed by the middleware library, a good fallback is one where you are expected to provide a pair of missing functions for `alloc()` and `free()`, you can override weakly declared functions with your own, or lastly, you can compile the library from source and are able to provide a `#define` macro used throughout for allocation. None of these methods incur per-allocation function call overhead because they are resolved by the compiler or linker. In this method, you have the option of forwarding allocations to the main heap (refrain!), or declaring a special heap that is dedicated to the subsystem using your choice of allocation strategy. Some middleware provide only a method for registering allocation callbacks. It is a common pet peeve to unnecessarily waste CPU cycles, and this is one minor gripe I have about otherwise solid offerings.

The worst possible situation is a library that is littered with direct calls to `new/malloc` and `delete/free`. This provides you no simple means to encapsulate the system's resources and no simple way to measure or limit its consumption.

1.4 Mass Storage I/O Access

Accessing optical media is very slow, and seeking—even on hard drives—can dominate load times. While certain kinds of middleware need to access the file system, particularly sound systems that stream music or asynchronous streaming systems that load game assets in the background, most do not need direct access to the physical media API.

These exceptions aside, middleware generally does require access to some assets, but it should never do so by directly requesting them from the underlying system API. A good middleware library provides explicit hooks for the developer to overload file and data requests easily so they can be funneled through any custom file system (e.g., WAD or pack files) in which the developer may have chosen to store data or redirect the file request to different hardware.

The most flexible libraries do not attempt to treat data as files or streams at all, rather they deal exclusively with bulk memory buffers and leave resource acquisition firmly in the hands of developers. This approach simplifies error handling and load time optimization, and it is more rapidly deployed to new hardware with dependable results.

The worst middleware libraries (I've seen this frequently in open source code) assume that a POSIX file system is always available, or one like it, and depend directly on C Runtime Library calls such as `FILE` and `fopen()`.

Another misguided attempt to address file system abstraction is the invasive extension of file streams by providing an abstract interface for a reader/writer class that the user provides. A derived instance of this interface grabs data via a virtual function one byte at a time, or even in small fixed sized blocks. This maps very poorly to real-world data and performance characteristics, and it should be avoided at all costs.

1.5 Logging

There will be times when expectations will not be met inside a middleware library. A good library handles warnings in a uniform way and has a way to integrate them into your existing log system. Better libraries will have some kind of trivial verbosity setting that ranges from noisy to absolutely silent and will preferably compile out the strings and error checking entirely. Noisy verbosity settings that spew plenty of details about the data being fed in affords developers a sense of trust that the library has been integrated correctly and its files are being read properly. However, if you cannot compile this out entirely, it comes at a cost of memory and CPU performance that is unacceptable in final shipping builds. Some middleware comes with release and debug libraries for this reason alone.

The best architected middleware products have a simple method for hooking a logging output callback so the logs can be integrated into the game's existing reporting system. Beware any library that blithely calls `printf()`, since this function is relatively expensive and may not even have a standard output pipe connected, providing no means to alert the user.

1.6 Error Handling

The best middleware will not have error conditions at all because they always work. I hold out hope that such a thing exists. Meanwhile back on Earth, errors are inevitable, and the handling of them is an important trait when determining whether some software can operate in your project environment. There are different schools of thought about when an error is always fatal, always recoverable, or in the gray area in between. Middleware libraries that give you no choice in the matter, no way to override the handling of errors, tends to be graded downward.

As console developers know, patching a game is often impossible, and some bugs are virtually impossible to reproduce or track down. Sometimes "heroic measures" are called for to guarantee the game does not crash even when some subsystem experiences complete and utter failure. Every piece of software should have some capacity to forward serious errors through a handler, where in a fit of desperation you can silence the sound system, clear the screen and print "You Win!" before hanging, or at least reboot. Steer clear of anything that appears to use `exit()`, `abort()`, or even naked `assert()` calls.

1.7 Stability and Performance Consistency

Middleware should be stable. After all, the main reason developers don't write something themselves is the time required to write and debug it. Libraries that are unable to report (and attempt to recover) from common trivial errors are fragile, and will be often cursed by developers. The best middleware will never crash, will fix garbage inputs or ignore them, and will leave plenty of debugging breadcrumbs for programmers to follow while tracking down the problem. Top tools and engines can sustain significant data corruption and missing files without crashing outright, which can end up impeding an entire team's progress while dealing with the problem.

Every project has different expectations when it comes to performance, so it's up to you to judge for yourself what is acceptable in absolute terms. However, every middleware library that you consider usable should also have a consistent memory and CPU performance profile. A stable frame-to-frame memory footprint is essential for shipping games on time. Occasional spikes in the frame rate can be hard to track down, too, and severely degrade the game experience. Good middleware should be a rock when instrumented with a profiler in every situation.

A recent project I worked on had a single frame memory spike of 2 MB and 100

ms. after tracking it down, it was due to a minor change in how a level script was written. A more consistent virtual machine would have limited the number of instructions it would execute or limit execution to a time slice. A more stable library would have kept a closer watch on its memory usage. Anecdotal evidence is sometimes all you can get here until you run afoul with personal experience, at the worst possible time. Ask around.

1.8 Custom Profiling Tools

Whenever considering a package for inclusion, I am impressed with tools that reduce uncertainty near the end of a project. These include any system that comes with a monitoring API, or better yet, a profiler of some sort, that cuts days of potential programmer time near the end of the game when memory is tight, CPU time is scarce, and nobody knows where it's going. Being able to immediately pull out a tool that can instrument a part of the game helps to rapidly narrow down the search. Gaining visibility into content is typically hard, so any help your middleware provides is tremendous.

1.9 Customer Support

Commercial middleware typically offers integration specialists, telephone support, direct email support, and forums or mailing lists. Sometimes, this is exactly what a team needs to move forward, especially when they reach a major stumbling block. The whole reason for middleware is to reduce risk and uncertainty, which customer support does. Middleware that comes with no expectation of support is useless. I have discarded more than a few great and promising technologies simply because the author was unavailable to answer a couple of questions.

Watch for fast response times in forums and dedicated personnel who answer questions on the phone or in email, and be prepared to send your integration code to them when they ask for it. (That's all the better reason to keep it tidy.)

1.10 Demands on the Maintainers

Programmers are busy, expensive resources. Middleware that requires a lot of effort and attention to integrate well with your flavor of build system is going to be kicked to the curb very quickly. Ideally, you simply add a library to the build process, `#include` a header file, and call a few functions. Ta-da! It's integrated.

Sometimes, the middleware is more invasive, and requires various `#define` macros to be set to configure it. Or perhaps it needs to be integrated directly into your project and compiled with your game. Additionally, some middleware has external dependencies that must be present for it to compile. Worse still, it may require introducing a new tool into the build process that may not work well with the build system. Turnkey systems are clearly preferable. I look for middleware that comes with GCC and Microsoft Visual Studio project files, but with extremely basic project configurations. This proves that the compilers I care about can handle the code, and that I can throw away the project files and integrate them my own way after an initial build using the provided project file.

1.11 Source Code Availability

Eventually, you may need to debug into the source code of a middleware product you plan to integrate. If it is a proprietary, closed-source product without a source code option at a reasonable price, look for alternatives. While certain "industry secret" parts of libraries may be binary-only, vendors know that source code is expected and will

often provide it with a license. Those who don't provide source typically claim that their customer support obviates that need. While this may be true, the moment there's an issue that customer support can't solve, but source code could, is the moment we start searching for a replacement. Again, a big reason for using middleware is that it's proven and stable, so source code should not really be necessary. But the availability is still important should the need arise.

1.12 Quality of Source Code

Having source code does not always mean you have the ability to make meaningful changes. The best middleware has excellent documentation that is auto-generated from source on every new code drop. It will have a familiar and consistent bracing scheme, some kind of naming convention for functions and variables, and ideally will live inside a brief namespace.

Take a moment to peruse the important header files. Inspect for `#define` macros for language keywords or common functions such as `new`, `min`, and `max`, or in fact any macro that escapes the scope of the header which could cause issues elsewhere. Use an archive-inspection utility such as `dumpbin` (in Windows) or `nm` (in Linux) to verify that the only exported symbols the middleware library defines are in a consistent namespace to avoid conflicts with other libraries or your own code.

Questionable middleware will be littered with `#pragma` statements, will disable errors, will reduce the warning level, etc. Scrutinize this heavily. These statements being in header files can harm the quality of your code and may cause some files to stop compiling simply by including them.

1.13 Platform Portability

Certain kinds of middleware are likely to be platform independent, but even if you have the source code there may be lurking byte ordering (endianness) issues. Inspect the file and network stream handling for endian swapping code or perhaps different asset building tools per-platform that sets up data into native formats.

Multithreading is very common in games today. Some middleware libraries predate this shift. Determine how thread locking is handled, if the code is thread-safe, and whether the thread controls are easy to override with a different platform's interface. Be aware that any library that does nothing with threading is, by its very nature, likely to be unsafe to use in a threaded environment. Factor in time for making the library thread-safe, or at least limiting its use to a single, specific thread at all times.

Also consider whether your game may be ported to a less-capable hardware platform at some point. If it might, determine what features are unique to this middleware offering that would make porting to a different library particularly difficult if this library does not exist for lower-end platforms.

1.14 Licensing Requirements

I am not a lawyer, and this is not legal advice. Consult a lawyer of your own to answer specific questions, and read every license personally. Also note, most publishers and even some hardware manufacturers have strict regulations about what open source licenses are allowed in products you develop. Clear all open source licenses through their legal departments before integrating one. What follows are my informed layman's opinions.

Every open source license that I know of springs into action as the result of

distributing software. What this means is, as long as you do not distribute the software outside your company, you can use whatever you like for applications used internally. Feel free to use any open source code in your internally-facing server technology or tools, but do not plan to distribute those programs.

Public domain code is completely harmless. Once you copy the code to your hard drive, you own it, and every modification you put into it is yours to keep. You may of course change the license to something else, release it back to the public domain, or keep it secret. It's yours.

The MIT, Zlib, etc., licenses appear to more or less disavow any responsibility for how you use their software. They do not require anything except a credit clause and that certain notices remain in the code. This is not a restrictive license with respect to retail use.

The LGPL license has some stipulations that require distribution of the source of the library (and possibly parts of your own application). Read this carefully before using in retail products.

The GPL license and many other similar licenses require complete code release on distribution if you integrate them into your products.

Frequently, commercial middleware licenses will require credit attributions, splash screens, an intro movie, etc. Be careful to adhere to these, and give credit where it's due—middleware authors helped make your game happen.

1.15 Cost

Every game has a different budget for middleware. Expect prices to vary widely from \$100 to \$750,000, depending on what kind of product you're looking at. Great middleware demands professional attention to detail, and when using a library

professionally, good support is important (but often costs more depending on your level of need).

When looking for a library that costs money, consider whether the cost is per-seat, per-game, per-platform, or if it has a maintenance license such as per-annum. Many middleware companies have special rates for digital download titles versus retail titles. Also, be willing to call up the account manager at your chosen middleware shop and try to sweet talk them into a discount. You might be surprised at how flexible they are if this is your first title with them—they know very well that middleware tends to become entrenched and expensive to remove, so it's good for them to get you hooked on their product. Try to work out multiple project deals, if you have that kind of leverage.

Some libraries, such as Zlib, have evolved to the point where there is plenty of support online in forums and sample code, and the number of bugs is approaching zero. When libraries calcify, the fact that they are free is not an issue. However, open source libraries are not free in general unless your time has no value. Take an estimate of how many man-weeks it will take to implement the various missing features listed above, assign a price to those man-weeks and compare that with the cost of the middleware library. If they're even close to parity, go with commercial middleware, simply because it's established, tested, supported, and best of all, you will free up programmers to work on actual game features instead of patching up someone else's pet project.

References

- [1] Kyle Wilson. "Opinion: Defining Good Middleware". *Gamasutra.com*.
http://www.gamasutra.com/php-bin/news_index.php?story=20406

2

The Game Asset Pipeline

Rémi Arnaud

Screampoint International

Highlights

The Game Asset Pipeline (a.k.a. the Content Pipeline, and hereafter, simply the Asset Pipeline) is an intricate mechanism that interfaces the artists to the game engine. Game teams have seen their artist/programmer ratio skyrocketing in order to cope with the quantities of high-quality assets on tight schedules. Making sure the assets are delivered in the right form at the right place is an important part of managing the creation process so that artists and designers can create, preview, add, and tweak content as easily and swiftly as possible.

This gem is composed of the following sections:

- Asset pipeline overview: What is the role and composition of an asset pipeline?
- Asset pipeline design: What are the main goals and hurdles involved in designing an asset pipeline?
- Push or pull pipeline model: Approaching the problem from a different perspective to achieve a better design.
- COLLADA, a standard intermediate language for digital assets: Taking advantage of a standard and its broad availability to build an asset pipeline.

- OpenCOLLADA, a new open source framework based on SAX parsing and direct write technology.
- User content: Retargeting the asset pipeline to foster user content creation.

2.1 Asset Pipeline Overview

The asset pipeline is the path that all the game assets follow from their creation tool to their final in-game version. There are a lot of asset types needed in a game: models, materials, textures, animations, audio, and video to name a few. The asset pipeline takes care of obtaining the data from the tools used for their creation, and then optimizes, splits or merges, converts, and outputs the final data that can be used by the game engine. A typical asset pipeline is described in Figure 2.1 and explained in the following paragraphs.

Source Assets

Source assets are created by artists and designers. The tools used to create source assets are usually referenced as Digital Content Creation (DCC) tools. In general, source assets are created through a set of specialized tools: one for modeling, one for texture creation, one for gaming data, and one for level layout. Source assets should contain all the information necessary to build the game content. In other words, it should be possible to delete all the assets except for the source assets and still be able to rebuild the game content. Source assets are often kept in the tool-specific format, and they should be stored using a version control system such as Subversion [1] or Perforce [2]. Since source assets are rather large and often in binary format, be sure to use a version control system that can handle large binary files.

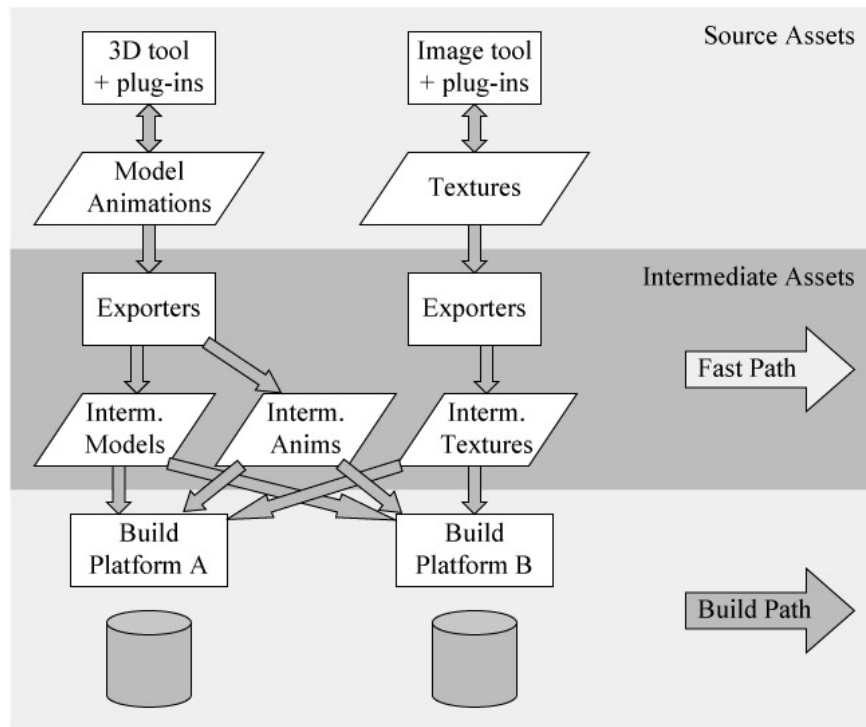


Figure 2.1: Asset Pipeline Components.

Final Assets

Final assets are optimized and packaged for each target platform. Often enough, the developer has no choice in what the final format has to be since the target platform, third party engine, or the publisher may impose a given format. There is no need for superfluous information in the final assets, so all the data is pruned to its required minimum. Just like source code, assets have to be compiled and optimized for each target. For instance, textures should to be stored in the format and size that the hardware can use immediately, as texture compression algorithms and parameters need to be optimized for the target texture hardware and display capabilities. Geometry data is also compiled for each target. Most platforms will only accept triangles, but the

number of triangles per batch, the sorting order of the vertices, the need for triangle strips, and the amount of data per vertex will have a big impact on the performance of the game engine. Limitations on the shader rendering capabilities of the hardware may also have an impact on the geometry since it may be necessary to create several geometries to enable the engine to use these in separate passes to create the desired effect.

Final assets also have to be optimized for game loading speed. Conversion of the data into hardware binary representation is one of the possible optimizations since this will save CPU time while loading, but it is not necessarily the main bottleneck to optimize since I/O (bandwidth or seeking) may actually be more problematic than CPU performance (parsing, decoding) nowadays. For really simple games, it may be enough to compact the data as much as possible and load it once at launch or at each change of level. But for more evolved games, the expectation is that the data will be continuously streamed while the player is progressing in the game to provide a better immersive experience. Having a good real-time data streaming capability built into the engine is one of the fundamental core technologies that will also help with providing a similar experience for the player on various platforms, as limitations in device memory size can be hidden by a local cache mechanism. Also, if the content is to be stored on a hardware format that has specific limitations (for example, a CD or DVD may have long seek times), it may be important to sort the data in the order in which it will be encountered in the game.

Build Process

The build process is the process that transforms all the source data and creates the optimized packaged data for each target platform. The build process should be automatic, as done with the game source code. The build should be optimized to only process the data that has changed from the previous build. Building the optimized data for an entire game can take hours, if not the entire day. It is especially painful at the end

of a project when there is a lot of tweaking that needs to occur in a short amount of time and the build time is the longest as the data repository is fully populated. Early in the process, little attention may be given to the optimization of the build process and the organization of the data, but the overall quality and success of the game will depend on the final editing that occurs when most of the assets have been created. In other words, a poorly designed build process is likely to directly impact the quality of the end product.

One of the major issues with optimizing the asset build process is the lack of information about dependencies. Since source assets are often stored in a tool-specific opaque format, it is not often possible to easily parse the data and recursively collect the dependency information. The need for dependency information and the direct extraction from the source has been in place for many years in source code development, but unfortunately, such a process has not yet matured for assets and will require the game engine developers to actively develop their own.

There are a few good sources of information about asset build systems, such as the book *The Game Asset Pipeline* [3] and the Microsoft XNA content pipeline [4]. We mention the latter in spite of the fact that XNA is designed to only target Microsoft platforms (PC and Xbox 360) and has several design limitations, such as relying on file extensions to separate the data types and imposing the build information to be embedded in the source code rather than as descriptive data [5].

Manifest

One very common issue with game development is that it quickly becomes impossible to know what assets are needed, what assets depend on others, and what assets are no longer necessary. Often data is piled up into a shared directory during development, and the game works fine since all the data referenced by the scripts or other assets are available. The problem is that there is no easy way to know which assets are really necessary, so packaging a demo in the middle of a project often results in

several gigabytes of compressed data, making it hard to distribute. Manually maintaining a manifest is easier said than done, as it requires imposing a strong policy on artists and designers to make sure they add all the assets referenced into the manifest and remove the ones not in use. In practice, there is no hope for maintaining a manifest manually, so it is very necessary to create an automatic system that extracts the manifest from the game's top-level description (i.e., level design files). Having access to external references stored inside the assets, such as scripts, textures, models, animation, or other references becomes an obvious need in order to be able to automatically create the manifest, so one must be wary of using opaque formats in the asset pipeline. The good news is that this information can be used to create a dependency graph enabling the build process to work only on the files that are necessary for a given level. The dependency graph also provides an understanding of what assets need to be rebuilt when they depend on assets that have been changed.

Fast Path

The fast path is the ability for assets to be loaded into the game engine without going through the full build process. This shortcut is very important to provide for better artist efficiency. When using the fast path, the artist invokes a minimal export mechanism that enables a faster iteration. Note that an artist is often working on a particular subset of the assets, and only the data for those assets will follow the fast path; the other elements of the scene can still be loaded from the final build path. This means the engine needs to allow for both full build and fast path assets to be active simultaneously during production. The fast path loader can later be pruned from the engine unless the developer wants to keep this capability as a feature to modders (see *Section 2.6*). The fast path is an optimization of the overall production process, which may be very important to the success of delivering the game within time and budget constraints.

Intermediate Assets

Intermediate assets represent the data in between the source and the final form. Intermediate assets are created using an exporter from the DCC tool and also represent the data that has been processed along the build process just before final transformation into the final asset packaging. The intermediate asset format should be very easy to read, parse, and extend. Intermediate assets should contain all the information that may be necessary down the pipeline, and be lossless wherever possible ^[1]. There is no need to prune and over-optimize early; it is much more efficient and automatic to discard data further down in the pipeline rather than having to re-export all the source assets each time additional information is required. Plain text, or structured XML files [6] are recommended formats for most intermediate assets; this provides for human readability and occasional hand editing in addition to taking advantage of numerous libraries capable of handling standard XML format. Especially useful is the built-in capability from languages such as C#, Python, Java, and others to be able to directly load an XML document and provide easy programmatic access and search functions to the embedded document object model (DOM).

Ideally, one single flexible intermediate format should be used to store the data during the entire transformation process so that the same format and I/O code can be used throughout the content pipeline and provide the flexibility to change the build process steps and their execution order. This means the format needs to store the data in a representation as close as possible to the representation in the DCC tool—in order to simplify the export process and ensure the least data loss as possible—as well as convert the data to a format as close as possible to the final encoding required by the target platforms. This involves a bit of additional complexity when designing the intermediate format, but ultimately, it is a big win to avoid overloading the export process with early transformations at the cost of export time and data loss. As shown in Figure 2.2, asset processing modules can apply transformations directly to intermediate

assets, such as triangulation, texture processing, and level-of-detail calculation.

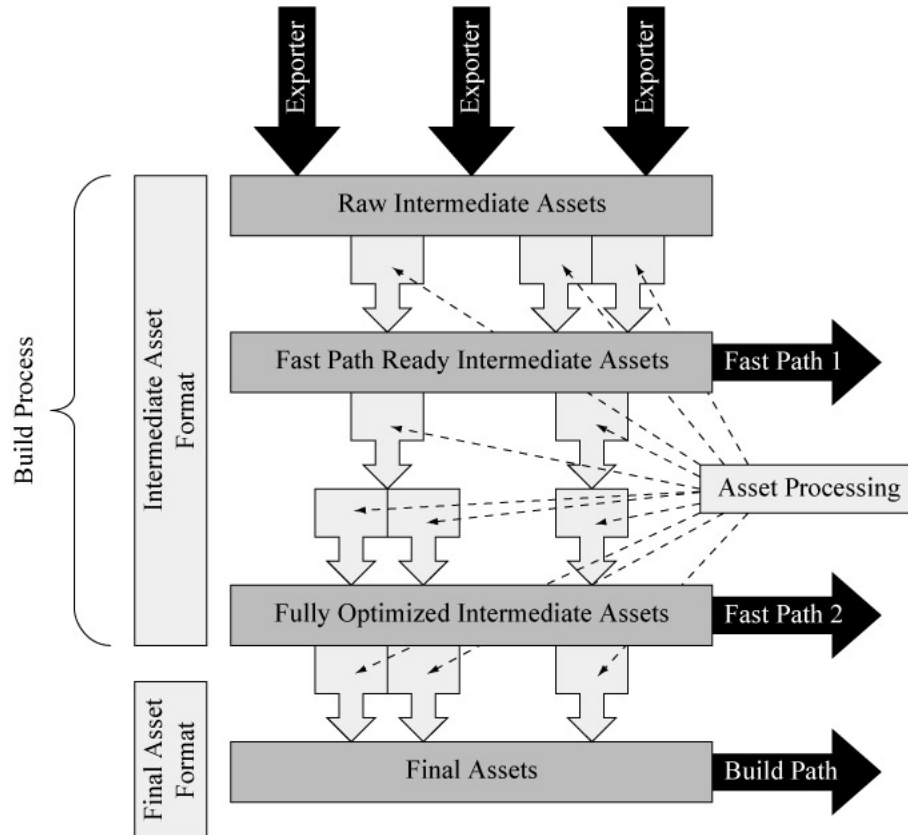


Figure 2.2: Intermediate assets along the build process.

The Pipeline

A combination of processing modules compose the build pipeline. Intermediate assets can be probed at any point in the pipeline since they are available in the same format throughout. Although not fundamentally necessary, it is possible to create a specific tool with a user interface that enables the creation of a build process by simply connecting asset processing modules to each other [7]. This design also enables the

build process to be more generic in order to handle data from a variety of sources and tools, as well as being configurable to support existing and future target platforms, by combining or configuring a set of asset processing modules differently.

There would be no need to use different formatting for source and intermediate assets if one single format could serve both purposes and is recognized by the DCC tools. This does not mean the data will stay untransformed through the build process; it means the same formatting can be used to store the data at all stages of the transformation. For instance, there exist hundreds of image file types/formats [8], but unfortunately, none seems able to offer all the properties to be both source and intermediate format. Ideally, we would have a format that can store uncompressed and/or lossless compression pixels for the source form. Some images need to be stored in higher resolutions than the usual 8 bits per component, so the format should allow for 16 bits, or even 24 bits per component. More and more often, high dynamic range (HDR) storage is necessary, for which an open format has been made available: OpenEXR [9]. On the other end, the intermediate format should be able to represent the data as close as possible to the target platform format, but the requirements are all over the spectrum to the point that it is almost impossible. For instance, final texture data has to represent mipmaps, cube maps, volume maps, height maps, normal maps, and texture arrays, and it has to deal with memory alignment, swizzling, tiling, and hardware compression (DXT). The D3DX10 format from Microsoft [10] is probably the closest to the needs for intermediate image format. It includes the DirectDraw Surface (DDS) description [11] as well as PNG [12], IFF [13], and BMP [14]. Moreover, it can also be used, at least on the Microsoft platform, as a final format. The problem is that there may not be an export and import workflow for this format in the image creation tool, forcing the use of two separate formats for the source and intermediate image assets.

In addition to the conversion from the source format to the intermediate format,

images also have to be made available to the DCC tool, which may require yet another format conversion. A very popular tool to create images is Adobe Photoshop, whose PSD native format can store a lot of types of data. Hopefully, main DCC tools can directly load PSD images and may even take advantage of layers and other additional information. Interestingly, Adobe has recently made available the open source Generic Image Library [15], which aims at simplifying and optimizing the processing of image assets, but unfortunately, it does not provide I/O support for Adobe's own PSD format!

In the past, game engine developers had to create their own intermediate format and write their own exporter as a plug-in for each vendor-specific tool in use by artists, which can be a daunting task, especially when tracking DCC SDK issues. Luckily, this is no longer a necessity, and engine developers should instead take advantage of a standard intermediate format and available source code [16], as discussed in Section 2.4.

^[1]Intermediate assets stored in an open and well-documented format are also very important for archival of the game assets. Here's an interesting story: A large studio shipped a game based on a new IP and cleverly stored all the source assets, source code, and build processes. It took many years, but eventually the sequel was ordered. The problem was that the source assets were not recognized by the new version of the DCC tools. Cleverly, they had also stored the tools used to create the assets. The problem was that there was no way to get a license server working for those old tools. Intermediate assets were stored in a binary opaque format, so no help there. All the assets had to be created again. Using a text/XML documented intermediate format in the archive would have saved a lot of time and money.

2.2 Asset Pipeline Design

From the end-user perspective, the asset pipeline should be transparent. In fact, the ultimate reward for the developer of the asset pipeline occurs when artists and game

designers do not know about the existence of the pipeline and feel there is nothing in between their creation tools and the game engine. The most important piece of the asset pipeline from the user's point of view is the game engine editor. It is the tool that provides the asset integration interface into the game engine. The editor is built on top of the same engine core technology as the game itself and provides artists, designers, and engineers the necessary WYSIWYG (What You See Is What You Get) environment for putting together the assets and addressing design or engine issues rapidly.

Game Engine Editor

There are two possible workflows to consider for the game engine editor:

1. The editor is an advanced viewer, enabling the user to check the assets within the engine and have control over the camera position, lighting conditions, and which assets are displayed. In this design, the assets are modified and edited only in the DCC tools and are exported whenever there is a change.
2. Some of the assets can be edited using the editor. This alternate design creates a more complex asset pipeline workflow, as the editor is effectively creating source assets that need to be stored properly for the build process.

The two main concerns when designing a content pipeline are efficiency and flexibility [17]. Efficiency is required in order for the team building the game to be able to deliver great content on time, and flexibility is required to be able to adapt the pipeline to the evolving needs of the team. Unfortunately, these two priorities are somewhat in opposition to each other, so there isn't a single solution that can satisfy all projects. Instead, a choice of compromises defining the requirements for the design of the asset pipeline is needed.

Therefore, the game engine editor will offer a combination of view-only and editing capabilities based on the specifics of a project. The question is where to draw the line. Obviously, it is not practical to rewrite all the editing functionality of all the DCC

tools in the editor. This would be a major task and change the focus of the development from creating a game to creating DCC tools, which is a different business all together. Some have tried to go the other way around, writing the game editor as a plug-in to a DCC tool. But it does not work either, as this requires writing (for example) an image editor, audio editor, script editor, AI tool, and level editor inside a 3D package, and it locks the tool chain to the extension capabilities of the tool SDK. The right compromise is to write only the editing features in the game editor that are either very specific to the game itself or provide a very large efficiency improvement over off-the-shelf tools. In fact, a game team should always be on the lookout for the availability of new tools that could improve their efficiency. The asset pipeline should be designed to provide enough flexibility to enable the inclusion of new tools, which can be a delicate process throughout the course of a project.

The other main point to consider is robustness. The asset pipeline is the backbone of the project and the team. The project is at a standstill when the pipeline is broken, and everyone on the team is idling while they wait for the pipeline to be fixed. A solid design and the proper documentation telling a team how to use and extend the asset pipeline is the key to its robustness.

Efficiency, flexibility, and robustness are all main goals of the intermediate asset format. The primary advantage of providing a step between the source and the final asset format is decoupling engine development and asset creation. If the final assets are to be created directly while exporting, most changes in the engine would require a synchronized change in the exporters, which would require re-exporting all the assets. It is much more efficient to cache the intermediate assets and provide the necessary changes to the fast path and the final steps of the build than it is to load all the assets back into the DCC tools and re-export. Doing so may require very expensive and time-consuming manual operation since most DCC tools are not easily or efficiently driven from the command line. A well designed intermediate format is not likely to change

much during the course of a project. If correctly cached and the source-asset-dependency tracked, the final assets can be recreated from the intermediate assets automatically, mimicking what is already true today for source code (when building the executable does not need to rebuild all the intermediate object code). Having an intermediate asset format in an easily readable format provides an easier way for game engine programmers to debug and experiment.

An advantage of the intermediate asset design over the object code is that it can provide for cross-platform asset creation. The build process can create final assets for several platforms from the intermediate assets, favoring a game design philosophy where source assets are created at very high resolution and the build process takes care of down-scaling the assets for the various targets. Automatically converting high-resolution assets is not always possible, but it is much more flexible and efficient than having to create source assets for each target. An example of this idea used in most pipelines nowadays is to create high-density polygon models, and then create a normal map texture associated with a lower density model, where the number of polygons and texture size can be adapted to the different targets as well as provide a means for level of detail management. Looking into procedural definitions of assets is a good way to provide for better automatic adaptability, although the tools available in that space are not as mature as the traditional DCC methods that are easier for artist to master.

Processing the high-density or higher-level abstraction assets into final assets can be a complex and resource-intensive process that is better done with an independent build system that can be optimized taking advantage of multi-core CPUs and number crunching APIs for the GPU, such as CUDA. More and more tool and middleware providers have started to take advantage of this technique to provide faster processing. A good design goal is to progressively move the asset processing into the engine itself, taking advantage of the ever growing available processing power. This provides for a sustainable technology path that will convert compute power into more flexible and

efficient content creation.

2.3 Push or Pull Pipeline Model

It is really tempting to directly modify an intermediate asset without editing the source asset and re-exporting. When pressed for time, it seems like the right thing to do, but most of the time, this is not such a good idea since intermediate assets get overwritten next time the source is modified and exported.

The ideal solution would be to be able to make changes at any point in the pipeline and make sure that those changes are reported back in the source data. One way to do this is to be able to import the intermediate assets back into the DCC tool so the modifications can be merged back into the source assets. But there is a slew of data in the source assets that are not reproduced in the intermediate data, such as construction history and tool widget layout. Additionally, the intermediate asset may have been processed so that the original form of the data is lost. Therefore, reimporting the intermediate assets should be done with the idea that changes will be merged into the source. Unfortunately, unlike source code development where diff and merge tools are common and allow for concurrent development, detecting and merging changes is not a common feature of DCC tools. This lack of merge tools in the asset workflow has also directly impacted the way content versioning systems are used. When dealing with source code, it is customary for several programmers to make changes to the same source files and use merging tools to consolidate the changes later. But the lack of such tools for content forces the use of lock mechanisms, allowing only one person to make changes to a given source asset at a time. This in turn has forced the source assets to be split in a large number of files in order to enable concurrent development to take place, because if all the data was in a single file, only one artist would be able to work on the content at a time. This is robust, but not very flexible or efficient.

Traditionally, the asset pipeline has been designed as a *push* model, in which the user has to create the intermediate assets or final assets by invoking the build process and then load the result into the editor to see what the final result is. An improvement to this model is to use the game engine editor to *pull* the intermediate or final assets directly from the user interface. An example of this technique is implemented in the latest Torque 3D asset pipeline [18], where the game engine editor is actively listening to changes in the intermediate or final assets and automatically updates the content in the editor upon external changes. Another similar idea is described in *The All-Important Import Pipeline* [19], where the game engine editor provides the user with direct loading of the intermediate assets and automatically invokes the build process to create the final assets on demand. This enables the user to pull the intermediate assets directly, rather than having to invoke the build process manually. Those two ideas can be combined to listen for intermediate asset changes and automatically invoke the build process. This mechanism can be combined with the fast path loading mechanism to provide a better interactive WYSIWYG iterative process, while a background process creates the final assets and automatically replaces the "slow" content with "optimized" content whenever it is available, transparently to the user.

Those ideas are a step in the right direction, but do not yet address the problem of editing intermediate assets when the source asset should in fact be modified. The asset pipeline design illustrated in Figure 2.3 adds a Remote Control line where the user can select any content in the game engine editor, and provided that the intermediate asset stores the source asset dependency and associated DCC tool, the source asset editing tool is automatically launched and provided to the user so it can be changed and re-exported. Combined with the previous method where the engine is listening to intermediate asset changes and automatically invoking the build process, this provides for an efficient, flexible, and robust design of the asset pipeline. The asset pipeline is viewed as a pull model by the user. Intermediate assets are pulled into the editor, for

example, using an asset browser that can provide a quick preview of the assets categorized by type. The editor can automatically create the manifest and automatically build the dependency graph for the build by following the dependencies stored in the intermediate assets. The editor can be used to select one or several assets and invoke an edit command that will look into the intermediate asset information, or the user preferences, and invoke the preferred DCC tool to edit the correct source asset. The editor can also directly invoke the version control system to appropriately check out a local copy and check back in modified assets. A collection of generic models can be provided to serve as placeholders that can be inserted in the current game level for assets to be created later, enabling a task list for the art team to be generated automatically.

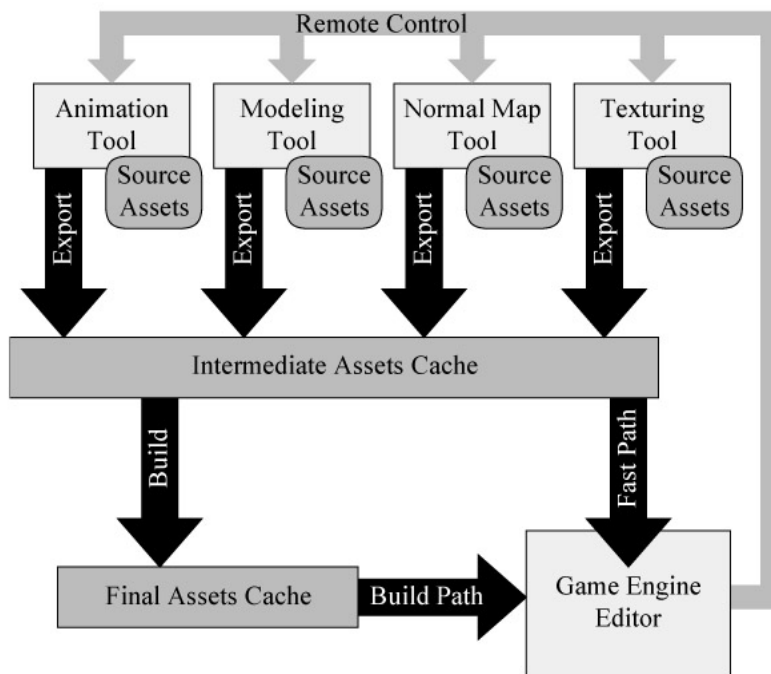


Figure 2.3: Game engine editor in control of the asset pipeline.

2.4 COLLADA, A Standard Intermediate Language

During SIGGRAPH 2004 [20], COLLADA was introduced as the first open Digital Asset Exchange (or .dae) specification targeting game developers. This initial specification was provided to the public after a year of work. Exactly a year earlier, during SIGGRAPH 2003, Sony Computer Entertainment was able to create a working group composed of key companies in the field to attempt the creation of a common intermediate language that would provide everything needed to represent the assets for advanced platforms such as the PlayStation 3. Companies such as Alias, Discreet, and Softimage that were competitors agreed to leave their weapons home and help with the design and prototyping. Game developers working at Digital Eclipse, Electronic Arts, Ubisoft, and Vicarious Visions liked the idea and offered to help.

Several iterations of the specification were completed, and thanks to the first developers that took the challenge to help resolve many of the issues in real use cases, the specification finally got to a stable state. In July 2005, the version 1.4 specification [21] was published as an open standard by the Khronos Group [22], the same group managing the OpenGL, OpenGL ES, OpenCL, and other well-known graphics standard specifications. The version 1.4 specification had minor fixes made, providing the 1.4.1 specification which, at the time of this publication, is the most popular implementation available.

During SIGGRAPH 2008, the version 1.5 version of the COLLADA specification was introduced [23], adding features from the CAD and GIS worlds such as inverse kinematics (IK), boundary representations (b-rep), geographic location, and mathematical representation of constraints using the MathML descriptive language. The version 1.4.x and 1.5.x specifications are not fully backwards compatible, which is not an issue since intermediate assets are to be re-exported from the source assets

anyway. As of today, the version 1.4.x format is the format supported by most applications, while version 1.5 support is limited to a few applications, mostly in the CAD space. Version 1.4.x and 1.5.x are to exist and be maintained in parallel. The new features introduced in version 1.5 will most likely never find their way into the version 1.4.x specification, though, so more and more tools are likely to provide both implementations. Loading version 1.4 or 1.5 should be transparent from the user's perspective anyway, since the document contains information about which version it is encoded with.

Since COLLADA is an open standard, many companies have been providing implementations. COLLADA was designed from the start to be a lossless intermediate language that applications can export and import. It is important for COLLADA to be a language, and in order to be useful, a language needs to be both spoken and understood. Also, to be able to validate and test an implementation, it is necessary to test an application with valid documents to load and then export in order to compare with the original data set to see if any data was lost during the process. This is the foundation of the official Khronos COLLADA conformance test that is available to Khronos Adopter members and provides a validation and certification process in order to provide the end-user with the assurance of a good quality implementation. The applications that have passed the test are authorized to display the conformance badge logo. Setting up an effective independent conformance test is no easy task, so before it is available and enforced there is some level of incompatibility due to the different possible human interpretation of the specification, but nothing that cannot be fixed by the end-user.

Since COLLADA provides for both import and export functionality, and since it is available for many tools, it has also been popular as an interchange format. The goal of an interchange format is to enable source assets to be transferred from one DCC tool to another DCC tool, which is definitely not the purpose of the asset pipeline. It turns

out that COLLADA is quite good as an interchange format, and has been providing for free, faster, and more accurate conversion of data between tools than some of the existing solutions.

The COLLADA 1.4.x feature list includes geometry, animation, skinning, morphing, materials, shaders, cameras, (rigid body) physics, and a transformation hierarchy. It is built upon the XML (Extensible Markup Language) standard for encoding documents electronically, so all COLLADA documents can be managed by any of the commercial or open-source XML tools. COLLADA is defined using the XML-Schema language [24] so that the COLLADA schema can be used by tools to validate the document or automatically create APIs in various languages as well as editing and presentation tools.

COLLADA is extensible in a very structured way. It is mandatory for an intermediate format to be extensible because the asset pipeline will need to encode data specific to the game engine or the game itself. The problem with extensions is how to design them so that the tools that do not recognize the extension can still import the intermediate assets and be able, when possible, to carry an extension forward. COLLADA provides several places in the schema where objects can be extended using the `<technique>` and `<extra>` elements associated with other elements in the specification. Since these elements are not permitted to be placed everywhere in the document, a COLLADA parser can be designed to recognize those extensions and either ignore the content, or better, keep the content in a string to pass the information through. Of course, if the tool recognizes the extension, it should use it! There are two ways to extend an object: either it is additional information augmenting the definition of an existing element, or it is an alternative definition of the common definition of the element. It is up to the developer to decide how to extend. In the case that the extension is made by substitution, it is a requirement to keep a common definition available that can be used by other tools as a place holder. For example, an engine using specific

geometry definitions, such as metaballs [25], can use the `<extra>` element to store the metaball information and keep a standard mesh geometry to store a generic mesh.

Let's have a look at some design principles that are applied to provide the flexibility necessary to represent data in a common language while still enabling it to be as close as possible to the specific DCC representation as well as enabling internal transformations to move it closer to the engine-specific representation. The remainder of this section is an overview of the COLLADA design principles. For more technically detailed information, the reader can refer to the COLLADA 1.4 and 1.5 specifications available on the Khronos website [22] and on the CD accompanying this book. These specifications and the COLLADA reference book [26] provide more details on the choices made when designing the standard.

The `<source>` Element

The `<source>` elements contain the raw data that is used by the assets. Like every other element, it has an `id` attribute that has to be unique in a valid document. A source contains a one-dimensional array of data of a specific type (ID, name, boolean, floating-point, or integer). Each element can have a name, which has to be a valid XML string of characters, but the name information is only used to store information relevant for humans and never used to reference an object in a document. Floats and integers can be expressed with any number of digits, providing for any level of accuracy needed. Despite popular belief, there is absolutely no loss of accuracy representing floating point numbers using decimal digits, provided that enough digits are used—9 for single precision and 17 for double precision [27]. The arrays themselves contain a `count` attribute that provides for easier memory allocation.

The `<technique_common>` element provides information about how the source should be accessed by the other elements defined by the specification. The `<technique>` element is where the extensions can be found; an element can carry extensions from

several tools at the same time since each tool provides a profile name for its technique.

Figure 2.4 shows that only one array can be in a `<source>` element by using a selector, and it shows that the `name` attribute is optional (by using dotted lines) and the `id` attribute is mandatory. All this information is stored in the XML schema available on the Khronos website, which can be automatically converted into a drawing or used to validate a document. Listing 2.1 shows what a source looks like in a document and that it is quite easy to parse.

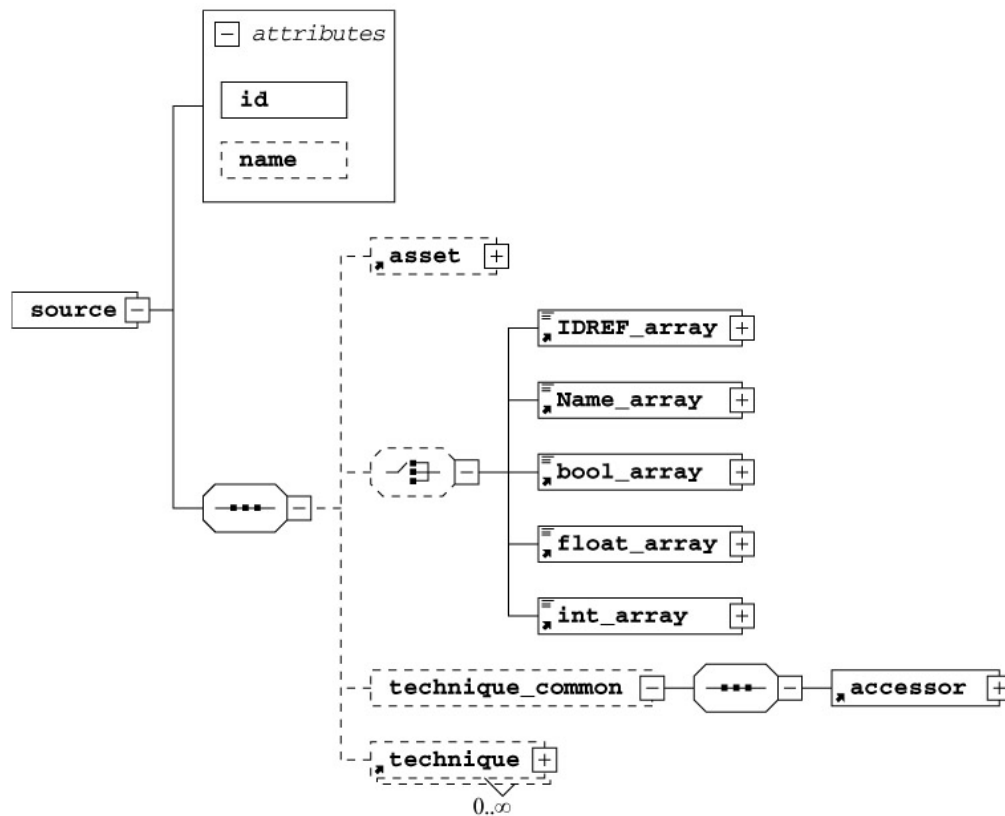


Figure 2.4: Definition of the `<source>` element.

Listing 2.1: A example <source> element and <accessor> element.

```
<source id="mesh1-geometry-position">
  <float_array id="mesh1-geometry-position-array" count="24">
    2518.1875 4074.512965 0. 2518.1875 0.
    ...
  </float_array>
  <technique_common>
    <accessor source="#mesh1-geometry-position-array"
      count="8" stride="3">
      <param name="X" type="float"/>
      <param name="Y" type="float"/>
      <param name="Z" type="float"/>
    </accessor>
  </technique_common>
</source>
```

The <accessor> Element

The <accessor> element, shown in Figure 2.5, is where the flexibility is built in. It enables us to organize the arrays in a format that is closer to either the tool or the target format, allowing the build process to convert from one to another. This provides better decoupling from the exporter's point of view since the exporter can export the data as is and then use the <accessor> element to explain how the data should be accessed. The `count` attribute tells how many elements can be accessed through the <accessor> element and what offset and stride is to be used, typically creating n -tuples of data. Then it describes one parameter for each element of the n -tuple, giving it a name and type. An example is shown in Listing 2.1.

If a <param> element is not given a name, that means the corresponding value is to be ignored. So a source array with 3-tuple values could actually be defining only a 2-tuple with one padding value. As you can see, it is possible within the same language to

represent a position array in many different ways. The astute reader will notice that the array element is optional in a `<source>` element, the reason being that an `<accessor>` element can reference an array stored in another `<source>` element, making it possible to reuse the same array of data in a different way through several `<accessor>` elements.

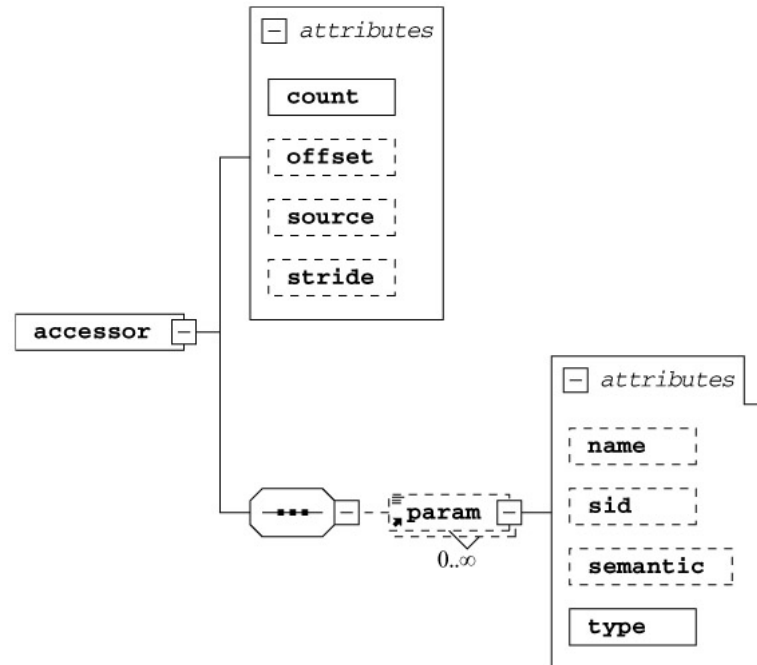


Figure 2.5: Definition of the `<accessor>` element.

Geometry and the `<mesh>` Element

Geometry is most often represented by a `<mesh>` element, although the `<convex_mesh>` element and the `<spline>` element are used for rigid body physics and 2D animation curves, respectively. The COLLADA 1.5 specification adds `<brep>` to the list of geometry types. A `<mesh>` element contains a collection of `<lines>`, `<linestrips>`, `<polygons>`, `<polylist>`, `<triangles>`, `<trifans>`, and

`<tristrips>` elements. Most likely, the mesh data found right after the export are polygons, and these are transformed into triangles closer to the end of the build pipeline.

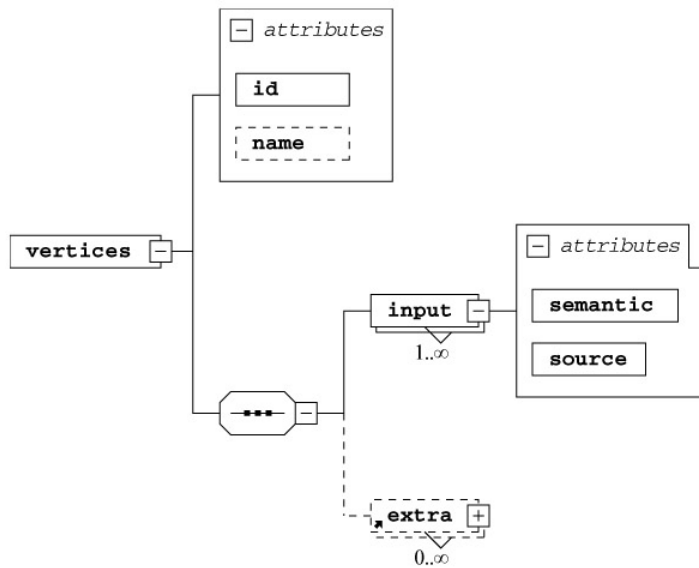


Figure 2.6: Definition of the `<vertices>` element.

A `<mesh>` element contains a mandatory `<vertices>` element, as shown in Figure 2.6. It has one mandatory input element that references a `<source>` element, meaning it uses the `<accessor>` element in the `<source>` element to access the data. One very interesting design feature is that there is no mention of the dimensionality of the vertex data in COLLADA. In other words, a vertex can have one, two, three, or any number of dimensions, though most content will be using 3D vertices with X, Y, and Z parameter names since transformations and positions are limited to 4D homogeneous coordinate space in the `<scene><node>` element. The `<input>` element associates one semantic with a source. Naturally, the one mandatory input is for the semantic POSITION, which

provides the position (e.g., x , xy , xyz , $xyzw$) of all the vertices used in the mesh primitives. A `<vertices>` element can contain as many `<input>` elements as necessary to attach additional data associated with each vertex. For example, it is customary to have a NORMAL stored per vertex, and it is also common in older systems to have a COLOR per vertex.

Referring to the `<triangles>` element in Figure 2.7 as an example of a `<mesh>` element sub-object, one can see it is also composed of a set of `<input>` elements that store the data associated with each primitive as opposed to each vertex as previously done in the `<vertices>` element. One of the `<input>` elements defines the semantic VERTEX, which references by index the POSITION defined in the `<mesh><vertices>` elements used for each primitive. Obviously, there are three vertices per element in a triangle list. The `<p>` element is therefore composed of a $P \times N$ set of indexes, where P is the count of primitives and N is the number of `<input>` elements defined in the primitive list.

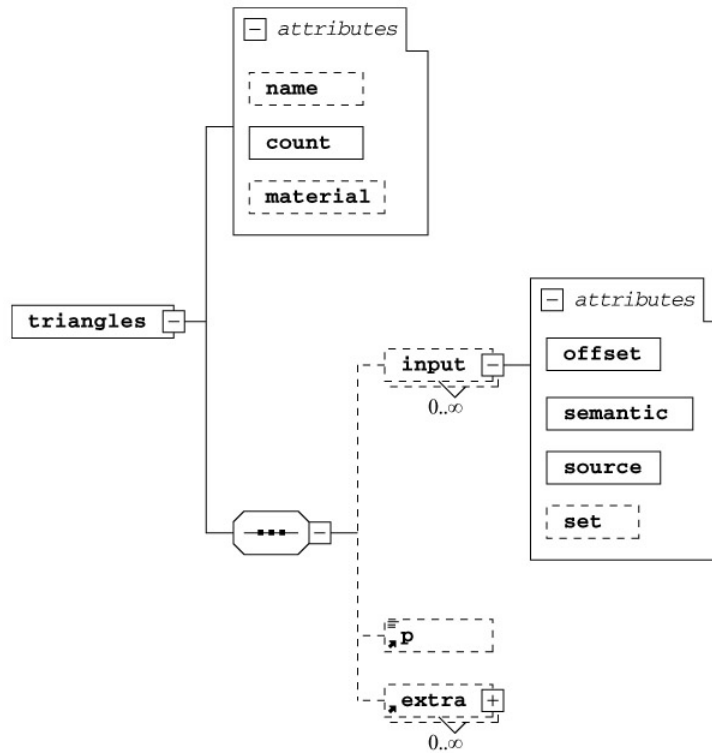


Figure 2.7: Definition of the <triangles> element.

Conclusion

To conclude this overview of the design of an intermediate asset format and the choices made for the COLLADA standard, it is worth mentioning a few additional high level concepts.

The most important and difficult design concept to keep in mind when designing an intermediate asset format is to try to avoid as much as possible a particular implementation or run-time. In other words, the data should be self-described so it can be transformed and used with any existing or future run-time. This is a really difficult design goal, and it's what occupied most of the design meetings in the COLLADA

working group, as the first draft of a feature is always close to how it will be used by the run-time or how it is created in the modeler. Making sure that the data is not described relative to one single usage model is very important, as this ensures the design is not made obsolete as technology is rapidly evolving or is limiting creativity by imposing a model that does not fit with yet-to-be-invented usage models. This one design point is the main difference between COLLADA and most other formats. This difference is obvious when comparing with Autodesk proprietary FBX interchange technology which is defined entirely through an API and specific usage model:

"The FBX file format is not documented. Applications use FBX SDK to import scene data from an FBX file or to export scene data to an FBX file."

— *FBX SDK Programmer's Guide*, page 6 [28].

Another important principle of design is the categorization of elements into `<library_xx>` element types that help with the organization of the data as well as enable document contents to be separated by type. Important is the distinction between the data definition and its utilization through instancing. This enables an element to be used many times without having to repeat its definition, which would cause the size of the intermediate and final assets to bloat tremendously. COLLADA enables some of the values of an element to be modified when instanced through the use of `<param>` elements, so it is possible to share most of the element definition and save a lot of space while still enabling source data changes to affect all instances, which comes in handy during production.

Last but not least, COLLADA takes advantage of URI technology [29], which enables elements to reference data within other documents for flexible organization, and it takes advantage of many different storage technologies as well. For instance, an external reference URI can be an HTTP request that is interpreted by a web server as a

database query, or it can be a simple reference to a file on the local storage device. One main issue that comes with utilizing formats that do not have a good external reference mechanism is that the intermediate assets all need to be grouped in one single document. Such a document can grow to an unmanageable size during the course of the project. Moreover, this means that *all* the data has to be imported and then exported by *all* of the tools used in the asset pipeline, which is a major limiting issue.

2.5 OpenCOLLADA

Given the open nature of COLLADA and its description using standard XML technology, there exist many commercial and open source options that can be used to interface with COLLADA documents. Some programming languages such as C# and Python provide libraries that can directly interface with XML data, providing the programmer with a Document Object Model (DOM) [30], a memory representation of the XML hierarchy that can be programmatically accessed. The libraries supplied with C++, the most used language in game engine development, do not provide this capability directly. There are several generic libraries that exist for loading and saving XML content, and there are a few commercial tools that can generate a specific access API created automatically within the XML schema, enabling the creation of an API specific to a particular document. A COLLADA DOM is available for C++ programmers, and it automatically generates a core API from the schema in addition to providing an API to help manage the objects once in memory. A DOM provides more than a loading and saving interface; it allows for the XML elements to exist in memory so they can be edited directly without having to create a separate representation [31].

More recently during SIGGRAPH 2009, NetAllied Systems announced the availability of OpenCOLLADA [32], which is the first open source implementation to

take advantage of SAX parsing and direct write technology, providing support for both versions 1.4 and 1.5. The project page is located on the web at <http://www.opencollada.org/>. The 3DS Max and Maya plug-ins built on this technology, as well as the source code for the framework, are available on that web page, and they are included on the CD accompanying this book for convenience. The reader is encouraged to check the web page for the latest updates.

SAX (Simple API for XML) is an alternative model to the DOM for parsing XML documents [33]. The main issue prevalent among most available libraries is that they load all of the XML data and create an in-memory representation of the data before a third copy of the data is made to create the object representation used by the program loading the assets. This problem is very common across all file-loading SDKs regardless of the format itself, and this issue is making it quite difficult to handle the very large assets that are becoming more and more common.

The same memory management issue exists when exporting content if a library is used to create a complete in-memory representation of the data to be exported before the data is written out. Workstation memory is commonly maxed out when a large model is loaded in a DCC tool, and as the export is invoked, the computer hangs as all memory contents are swapped out to make room for yet another copy of the same data. The same principle can be used when exporting content by writing out the data directory and avoiding the creation of another copy of the content.

The `DAE2Ogre` sample code shown in Listing 2.2 (and included on the accompanying CD) demonstrates how to take advantage of this technology. The idea is to associate a writer to the reader C++ object, where the reader is the COLLADA SAX parser, and the writer (in this case) is the Ogre engine-specific format. Since all data is not available in memory at once, it is not possible to follow references and expect to find the data in place. So instead, the SAX parser uses a `UniqueId` type for referencing.

Since the data is passed to the writer in the order that it appears in the COLLADA file, it might not be possible to resolve a reference immediately in the case of forward referenced data. To solve this problem, it is common to load the file twice. In the first pass, scene graph, material, and other data are gathered and stored. In the second pass, geometry, animation, and other data are handled. A SAX parser is a bit more complex than having all the data in memory, but thanks to the availability of OpenCOLLADA open source framework [34], it is not so difficult. As we have already seen, there is a big advantage in memory usage for such technology, which is mandatory when assets are very large, but it also turns into performance gain as we will observe.

Listing 2.1: This is a sample exporter from DAE2OgreOgreWriter.cpp.

```
OgreWriter::write()
{
    COLLADASaxFWL::Loader loader;
    COLLADAFW::Root root(&loader, this);

    // load and write scene
    mCurrentRun = SCENEGRAPH_RUN;
    root.loadDocument(mInputFile.toNativePath())

    // if there is no visual scene in the COLLADA file,
    // nothing to export here
    if (mVisualScene)
    {
        SceneGraphWriter sceneGraphWriter(this, *mVisualScene,
            mLibrayNodesList);
        sceneGraphWriter.write();
    }

    // load and write geometries
```



```
mCurrentRun = GEOMETRY_RUN;
root.loadDocument(mInputFile.toNativePath())
}
```

Tables 2.1 and 2.2 show the duration and memory consumption of an import operation and an export operation for a large scene in 3DS Max. The tables compare the open-source OpenCOLLADA technology and the Feeling Software open-source implementation using the DOM / intermediate memory model library FCollada [35]. Interestingly, the difference in performance between these two COLLADA implementations is observable regardless of the exact format used for storage, and the reader is encouraged to do some performance tests on large datasets to improve the efficiency of his asset pipeline, if needed.

Table 2.1: Import into 3DS Max using OpenCOLLADA for Max and Feeling's ColladaMax.

Boom.dae, 116 MB one mesh	OpenCOLLADA	ColladaMax
Time used for import	3.8 s	32.5 s
Max memory consumption during import	752 MB	784 MB
Memory consumption after import	444 MB	476 MB
Memory consumption after deleting scene	284 MB	332 MB

Table 2.2: Export from 3DS Max using OpenCOLLADA for Max and Feeling's ColladaMax.

Boom.max, 29 MB one mesh	OpenCOLLADA	ColladaMax
Time used for export	3.5 s	46.3 s

Table 2.2: Export from 3DS Max using OpenCOLLADA for Max and Feeling's ColladaMax.

Boom.max, 29 MB one mesh	OpenCOLLADA	ColladaMax
Max memory consumption during export	438 MB	623 MB
Memory consumption after export	418 MB	418 MB

2.6 User Content

Modding is a computer game community slang expression that is derived from the verb "modify", used particularly with regard to creating new or altered content. Modding was once regarded as a fringe activity, but is now encouraged since it extends the shelf life of games. Tools are provided to help the gaming community create additional content requiring the purchase of the game itself, which provides additional content and revenues at no additional cost for the developers. In fact, strong communities are developed and provide strong viral marketing, thereby creating additional stickiness and loyalty to the game developer. Content created by end users is referred to as *user content* or as *player content*.

In order for end-users to create content, the game developer has to provide access to a simplified or more robust version of the game editor, scripting, and content pipeline. Since end-users are not likely to have access to expensive DCC tools, it is important to provide an asset pipeline that can also take advantage of free tools such as Blender [36], Google SketchUp [37], or XSI Mod Tool [38]. Crymod is an example of a successful modding community for the Crytek games, which has its own portal on the web at <http://www.crymod.com/>. Crytek has partnered with Softimage (now owned by Autodesk) to offer a specialized version of their COLLADA exporter that provides a free-to-use professional grade tool with integrated asset pipeline to enhance the

creation of high quality user content. The modified exporter uses specific name semantics that are recognized by the Crytek engine and used to connect the created content to their physics and other game-specific entities. It should be noted that this asset pipeline (Mod Tool to COLLADA to CryEngine) was not used by Crytek to develop their own game, but was designed specifically for the modding community. But now that this tool chain has been developed, it has also found internal usage.

More and more user content can be found online in 3D content repositories. Google 3D Warehouse has been offering the capability for SketchUp users to upload content to this repository, allowing anyone to search and download the content in the COLLADA format. With the introduction of SketchUp 7.1 features providing free import and export of COLLADA, it now is possible for content created in any other tools to be uploaded into the warehouse to be shared with other users. Since the warehouse is connected to Google Earth, there are many models of existing buildings and ever increasing variety of content. Taking advantage of this content in the prototyping phase of a game is something that should be considered since it is a good shortcut for developing the gameplay before replacing the content with the real assets later on. Another user content website has taken a different approach: www.3dvia.com (a Dassault Systèmes company) enables users to upload the content encoded in many source formats and automatically run conversion tools on the server so that any uploaded content is made available in both 3DXML (a Dassault proprietary format) and COLLADA. At the time of this writing, more than 150,000 users and 15,000 models are active within this 3D user content community.

Some games go even further in taking advantage of end-user creativity. *Spore*, a game from Maxis/Electronic Arts, is a good example in which the user is provided content creation tools to make their own creatures, vehicles, and buildings to use in the game. The latest edition of *Spore* extends the principle to the creation of adventures, or mini-games. The motto of this type of game is that "the fun is in the tool", and *Spore*

users had a lot of fun, as it is reported that more than three million creatures have been created already. During SIGGRAPH 2009, Will Wright, creator of the *The Sims* and *Spore*, delivered a keynote [39] in which he announced an additional step in favor of user content. With the latest patch for *Spore*, users cannot only create content that enriches the game for everyone else, but can also export the creature from the game into the COLLADA format [40]. This small change opens up a world of possibilities for end-users. The creatures are exported with their skeleton and skin, as well as diffuse, specular, and bump maps. Already, many end-users have given freedom to their creatures from the *Spore* engine and used sophisticated rendering and materials to embellish their creations, which they then share with others. (For example, see the "Majestic Dragon" on the accompanying CD.) More sophisticated users are creating animations that have been posted on YouTube. These advanced users are now educating other users and trying out all the tools that can import COLLADA models. In the future, it is likely that short animated features will be created by end-users, but even more exciting will be the intersection of different gaming communities. It is possible that there are already *Spore* creatures fighting inside a CryMod!

The Future

Pandora's box is open, and there is no turning back. User content is growing, and the need for an easy-to-use asset pipeline for both end-users and professional content developers alike is growing. 3D is making its way toward becoming main stream media, just like audio and video in digital form are now commonly produced and consumed via mainstream media. It is expected that the need for a better asset pipeline is growing as 3D is becoming more pervasive. In particular, the consumer availability of native 3D display TVs and monitors [41], advanced shader-capable 3D accelerators in mobile devices [42], and native hardware-accelerated 3D rendering inside web browsers [43, 44] are also pushing the envelope.

References

- [1] *Subversion*. <http://subversion.tigris.org/>
- [2] *Perforce*. <http://www.perforce.com/>
- [3] Ben Carter. *The Game Asset Pipeline*. Charles River Media, 2004.
- [4] Microsoft. *XNA Game Studio 3.1*. 2009. <http://msdn.microsoft.com/en-us/library/bb203887.aspx>
- [5] Rémi Arnaud. COLLADA for XNA forum discussion and source code. 2008.
https://collada.org/public_forum/viewtopic.php?f=13&t=651 and
https://collada.org/public_forum/viewtopic.php?f=13&t=676
- [6] W3C. *Extensible Markup Language (XML) 1.0*, 5th ed. November 26, 2008.
<http://www.w3.org/TR/REC-xml/>
- [7] Sony Computer Entertainment. "COLLADA Refinery".
https://collada.org/mediawiki/index.php/COLLADA_Refinery
- [8] "Image file types". <http://www.fileinfo.com/filetypes/image>
- [9] Industrial Light & Magic. *OpenEXR*. <http://www.openexr.com/>
- [10] Microsoft. "D3DX10 image format". <http://msdn.microsoft.com/en-us/library/ee416748%28VS.85%29.aspx>
- [11] Microsoft. "DDS image format". <http://msdn.microsoft.com/en-us/library/ee418141%28VS.85%29.aspx>
- [12] Portable Network Graphics. "An Open, Extensible Image Format with Lossless Compression". <http://www.libpng.org/pub/png/>

- [13] IBM. "Standards and Specs: The Interchange File Format (IFF)". 1985.
<http://www.ibm.com/developerworks/power/library/pa-spec16/>
- [14] Microsoft Windows Bitmap Format.
<http://www.fileformat.info/format/bmp/spec/e27073c25463436f8a64fa789c886d9c/view.htm>
- [15] Adobe. "Open Source Generic Image Library (GIL)".
<http://opensource.adobe.com/wiki/display/gil/Generic+Image+Library>
- [16] Rémi Arnaud and Kathleen Maher. "COLLADA: Content Development Using an Open Standard". *Game Developer Magazine*, May 2007.
- [17] Noel Llopis. "Optimizing the Content Pipeline". *Game Developer Magazine*, April 2004.
- [18] GarageGames. "Effortless Art Pipeline using COLLADA".
<http://www.garagegames.com/products/torque-3d#feature-pipeline>
- [19] Rod Green. "The All-Important Import Pipeline". *Game Developer Magazine*, April 2009.
- [20] Mark Barnes and Rémi Arnaud. "SIGGRAPH 2004 COLLADA Tech Talk". 2004.
http://www.collada.org/public_forum/files/COLLADASiggraphTechTalkWebQuality.pdf
- [21] Sony Computer Entertainment. "COLLADA Approved by Khronos Group as Open Standard". July 29, 2005. <http://www.scei.co.jp/corporate/release/pdf/050729e.pdf>
- [22] The Khronos Group. "COLLADA—3D Asset Exchange Schema".
<http://khronos.org/collada/>
- [23] Khronos Group. "Khronos Releases COLLADA 1.5.0 Specification with New Automation, Kinematics, and Geospatial Functionality", August 5, 2008.
http://www.khronos.org/news/press/releases/khronos_releases_collada_150_specification_with_new_automation_kinematics_a/

[24] "The X3C XML Schema". 2001. <http://www.w3.org/XML/Schema>

[25] "Metaballs". 1999.

<http://www.siggraph.org/education/materials/HyperGraph/modeling/metaballs/metaballs.htm>

[26] Rémi Arnaud and Mark Barnes. *COLLADA: Sailing the Gulf of 3D Digital Content Creation*. AK Peters, 2006.

[27] David Goldberg. "What Every Computer Scientist Should Know About Floating-Point Arithmetic". 1991. http://docs.sun.com/source/806-3568/ncg_goldberg.html#812

[28] Autodesk. *FBX SDK Programmer's Guide*, 2009.

http://images.autodesk.com/adsk/files/fbx_sdk_programmers_guide_2010_2.pdf

[29] W3C. "Uniform Resource Identifier (URI): RFC 3986". 2005.

<http://www.ietf.org/rfc/rfc3986.txt>

[30] The XML Document Object Model from W3C, 2005. <http://www.w3.org/DOM/>

[31] COLLADA wiki. "COLLADA DOM Portal".

https://collada.org/mediawiki/index.php/Portal:COLLADA_DOM

[32] Khronos Group. "The Khronos Group Announces Significant COLLADA Momentum at SIGGRAPH 2009". 2009. <http://www.blendernation.com/the-khronos-group-announces-significant-collada-momentum-at-Siggraph2009/>

[33] Wikipedia. "Simple API for XML".

http://en.wikipedia.org/wiki/Simple_API_for_XML

[34] Netallied Systems GmBh. "OpenCOLLADA SDK".

<http://www.opencollada.org/faq.html>

[35] Feeling Software. COLLADA Support. http://www.feelingsoftware.com/en_US/3D-collada-tools/collada-tools.html

[36] Blender Foundation. *Blender*. <http://www.blender.org/>

[37] Google. *Google SketchUp*. <http://sketchup.google.com/>

[38] Autodesk. "Autodesk Softimage Mod Tool".
<http://usa.autodesk.com/adsk/servlet/pc/item?id=13571257&siteID=123112>

[39] Stephen Jacobs. "SIGGRAPH: Wright Talks Perception And 'Entertaining The Hive Mind'". *Gamasutra.com*, August 6, 2009. http://www.gamasutra.com/php-bin/news_index.php?story=24733

[40] Dan Moskowitz. "How To Export Spore Creatures to Maya".
<http://forum.spore.com/jforum/posts/list/37155.page>

[41] Marguerite Reardon. "3D is coming to a living room near you". *CES 2009*.
http://ces.cnet.com/8301-19167_1-10142957-100.html

[42] Apple. "OpenGL ES on iPhone OS".
http://developer.apple.com/iphone/library/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/OpenGLESontheiPhone/OpenGLESontheiPhone.html#//apple_ref/doc/uid/TP40008793-CH101-SW1

[43] Google. "O3D API". <http://code.google.com/apis/o3d/>

[44] Khronos Group. "Khronos Details WebGL Initiative to Bring Hardware-Accelerated 3D Graphics to the Internet". <http://www.khronos.org/news/press/releases/khronos-webgl-initiative-hardware-accelerated-3d-graphics-internet/>

3

Volumetric Representation of Virtual Environments

David Williams

Thermite3D

3.1 Introduction

The use of height maps as a mechanism for representing terrains is well established within computer graphics and gaming. Height maps are a conceptually simple representation, easy to visualize, and simple to create. Furthermore, there is a large body of research [19] into the manipulation and rendering of such data. However, there are also serious limitations that result from this rather simplistic representation, such as the inability to support caves and overhangs.

In this article, we take the concept of two-dimensional height maps and show how they can be extended to fully three-dimensional volumes. This is a representation that naturally and consistently handles the kind of geological structures mentioned previously. It also allows easy real-time modification, and as such can be used to create powerful terrain editors or unique game play opportunities.

To this end, the concept of volumetric environments has been successfully employed in several commercial games to date. The game *Worms 3D* found it to be an appropriate way to bring the highly destructible but two-dimensional levels from the

earlier *Worms* games into the third dimension ^[1]. The *Crysis* sandbox editor utilized voxels during terrain modeling, and these were turned into traditional static meshes for runtime use. And the upcoming game *MinerWars* uses the concept to allow players to dig through asteroids in real time.

Throughout this article we will consider the modeling and rendering of complex terrains to be the main application of the described technology. None the less, we do believe the technology can have direct application to manmade or otherwise artificial environments if appropriate game play mechanics and artistic styles are in place. In fact, the use of voxels for non-terrain environments has been a core research area of our own experimental Thermite3D game engine [15], upon which this gem is largely based.

Figure 3.1 shows a variety of environments that are represented using the volumetric approach described in this article.

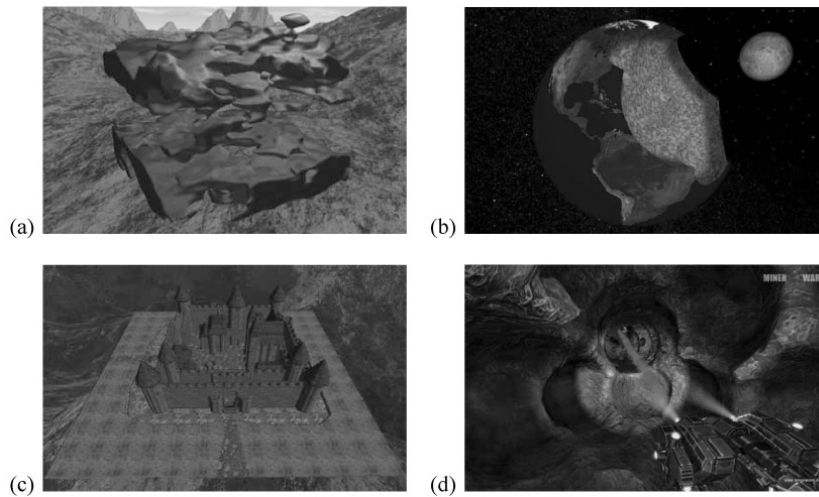


Figure 3.1: Volumetric representations can be used for many different types of environments. In (a) we see a complex terrain with two primary levels and numerous overhangs [5]. (Image courtesy of Thomas Schöps.) The Earth in (b) has been cut away to illustrate that the interior is also modeled [15]. Manmade structures with many different materials (c) can also be represented [15], while (d)

shows a mining ship inside an asteroid, destroying it in real time [8]. (Image courtesy of Keen Software House.)

^[1]Although based on the concepts described in this article, *Worms* actually uses a much lower resolution volume than that which we present here but allows for lattice deformations to achieve their desired artistic style.

3.2 Overview

The core data structure within our system is that of the *volume*. A volume is a regular three-dimensional grid of values, each of which is known as a *voxel*. Conceptually, this is analogous to the way a bitmap image is a regular two-dimensional grid of pixels. It is also possible to think of a volume as consisting of a number of two-dimensional slices stacked on top of each other. This is shown in Figure 3.2(a). We define the concept of a *cell* as being a group of eight neighboring voxels that form a cube, again as illustrated in Figure 3.2(b).

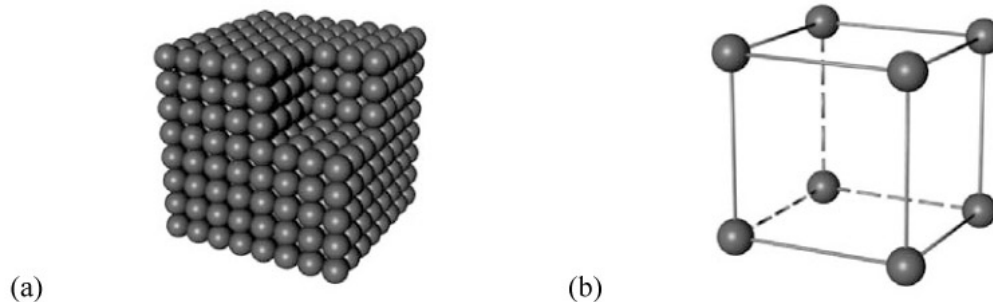


Figure 3.2: (a) This volume consists of an $8 \times 8 \times 8$ grid of voxels, though real volumes are considerably larger. The corner is cut away to show how voxels also model the interior of an object. (b) Each group of $2 \times 2 \times 2$ voxels forms a cell. Note that voxels and volumes are the only types that are stored explicitly within our system, as edges and cells are implicit constructs that we build by looking at a voxel's neighbors.

The volume is sized and positioned so as to cover the entire virtual environment that we wish to represent. Each voxel then encodes a representation of what exists at its location. The exact data that constitutes a voxel will be discussed shortly, but for now it can be considered to be a single bit indicating whether that location is solid material or empty space. Naturally this representation is very easy to modify in real time, because adding or removing material simply becomes a case of setting the values of the voxels. This is significantly easier than the complex CSG operations that might be required for other representations.

While the direct rendering of such volumes is an active research area [12], modern GPU hardware is highly tuned to the efficient rendering of triangle meshes. Therefore, although the volumetric representation is very useful for editing and deforming the environment, it is desirable to transform it into a triangle mesh for the purpose of visualization.

This process is known as *surface extraction*, and there are a number of algorithms that are able to perform it. The *Marching Cubes* algorithm [7] is one of the earliest and most widely used—it is popular due to its simplicity, speed of execution, and good locality of reference. Further developments have addressed ambiguities in the original algorithm, worked around (now expired) patent issues, or provided adaptive triangulation.

The Marching Cubes algorithm operates on a single cell at a time. For each corner of the cell, it classifies the voxel as being either inside or outside of the surface according to its value. This gives 256 possible combinations which can be grouped into the 18 equivalence classes illustrated in Figure 3.3. Each class represents the set of rotationally symmetric cases, and some classes include inverses as well. The last three classes in Figure 3.3 are additions to the original Marching Cubes algorithm that must be used instead of the inverted triangles in order to avoid holes.

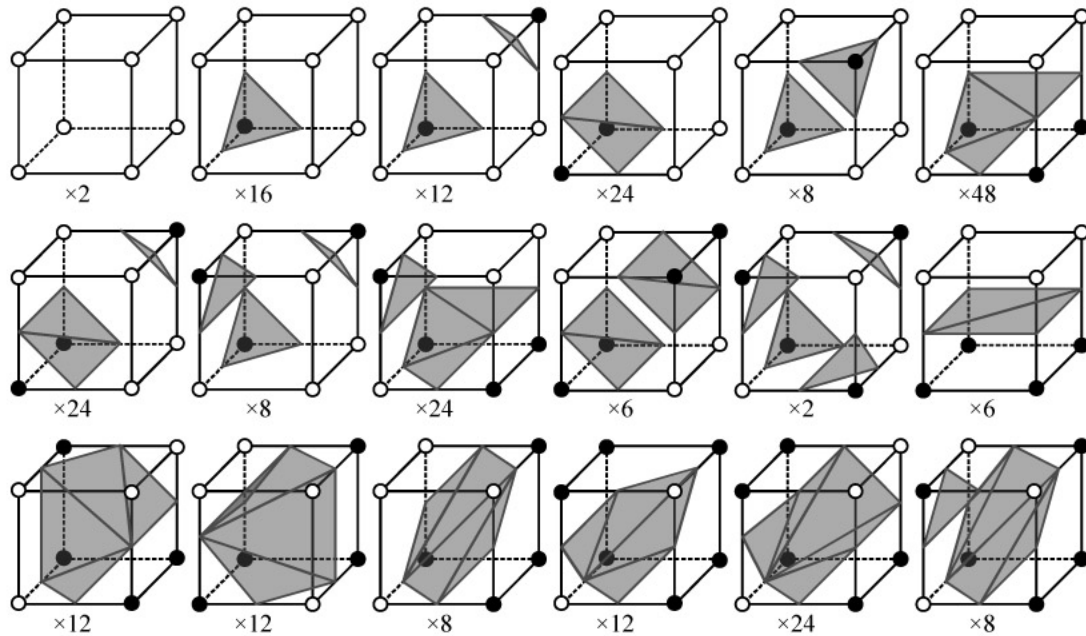


Figure 3.3: The set of triangles generated by the Marching Cubes algorithm for each of the 18 possible cell configurations. The numbers indicate how many times each configuration occurs. Solid circles represent voxels containing solid material, while hollow circles represent voxels containing empty space. In most cases the inverse configuration generates the same set of triangles, with the exception of the last three cases (which are inverses of earlier cases but with different triangles to avoid holes).

A lookup table is used to map the combination of voxels to a particular set of triangles that locally represents the surface. This process is applied to every cell in the volume (though cells with identical voxel values generate no triangles and can be trivially skipped) in order to reconstruct the complete surface. We will not be describing this process in much detail as it is already well covered by existing literature and numerous implementations are available online.

3.3 Data Structures

Having developed an understanding of the core principles, it is now possible to think carefully about the data structures involved in our volumetric representation.

Although each voxel can be represented by a simple inside/outside bit as described previously, in practice this does not provide much flexibility. Instead, our engine stores an 8-bit *material ID* for each voxel. A value of 0 represents empty space, while each of the 255 non-zero values represents a different material (rock, soil, wood, etc.). Naturally each of these materials will have a different visual appearance, but it is possible to attach different physical properties to them as well (perhaps some cannot be destroyed in-game, for example). This material ID will later be passed down the graphics pipeline for use in shading calculations.

Additionally, the use of a simple in/out decision when classifying the corners of a cell tends to lead to a mesh with a very jagged appearance. Depending on the application (and certainly in the case of terrain) it can be useful to replace this binary volume with a *density field*. In this case we assign each voxel a numerical value and we define the surface of our environment to be the set of all points that have a particular *isovalue*. When running the Marching Cubes algorithm on a given cell we classify each corner as being above or below the isovalue, and use linear interpolation to position any resulting vertices at the correct location along the edge. ^[2]

Using a density field rather than a binary volume means more control is afforded over the shape of the mesh. By modifying the voxel values such that the isovalue is not exactly halfway between them, the resulting vertex can be pushed closer to one voxel or the other. The consequence of this is that the resulting mesh tends to be a lot smoother.

Having defined the contents of a voxel we can now create a three dimensional grid of them to form our volume. A naive approach would be to store a simple three-

dimensional array of voxels such that they form a continuous layout in memory. However, given what we know so far, we can outline some desirable properties that we would like our volume data structure to exhibit:

- **Compression.** Using two bytes per voxel (for material ID and density) means that our volume will occupy $2 \times \text{width} \times \text{height} \times \text{depth}$ bytes of memory using the simple contiguous approach. This quickly becomes unacceptable for reasonably sized volumes, and so it is useful to instead use a data structure that exploits the high spatial coherence that volumes tend to exhibit.
- **Fast read access.** This is crucial firstly for implementing the surface extraction algorithm efficiently, and also for implementing picking and collision detection directly against the volume (rather than the extracted mesh).
- **Fast write access.** Allowing the real time modification of the volumes is one of the core requirements of our system. To do this we need to allow fast modification of voxels. While a structure such as an octree is likely to do very well at satisfying our compression requirement, it is likely to have a higher modification overhead as changes may have to be propagated up the tree.
- **Fast access to neighbors.** Accessing a voxel's neighbors is required when running the Marching Cubes algorithm (as this operates on a per-cell basis) and also for computing surface normals directly from the volume data (see Section 3.4). To achieve this we wish to make our data structure cache-friendly, such that voxels that are nearby spatially are also likely to be nearby in memory.

There is a large body of research on storage techniques that meet the requirements above, but we have chosen to use the approach presented by Grimm et al. [1]. Essentially the volume is broken down into a collection of cubic blocks, and the volume is represented as a list of reference-counted pointers to these blocks. See Figure 3.4.

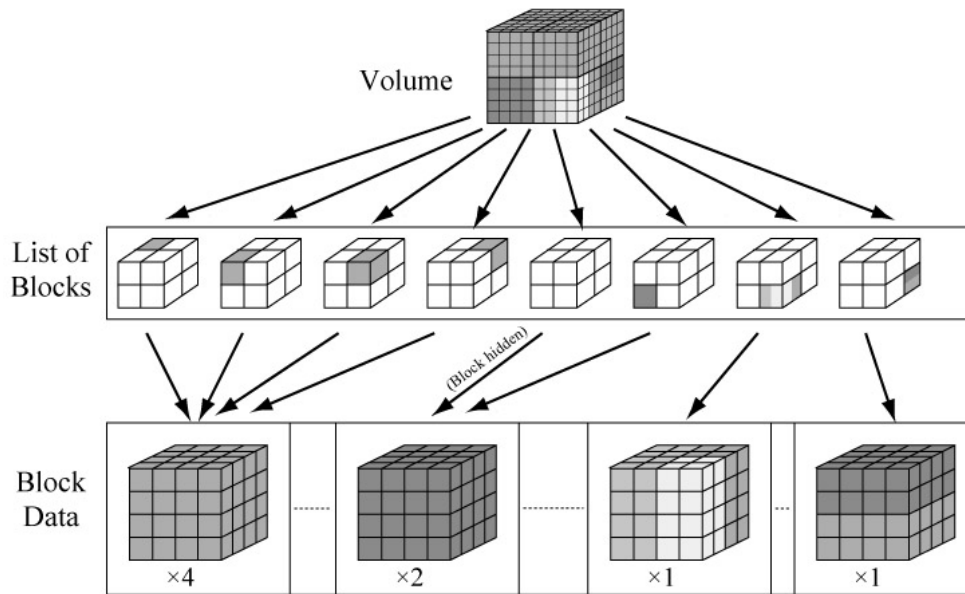


Figure 3.4: The volume with dimensions $8 \times 8 \times 8$ voxels at the top of the figure contains four different material IDs represented by colors (see figure on accompanying CD). It is split into 8 blocks, each of which have dimensions $4 \times 4 \times 4$ voxels. The four top blocks and the two lower left blocks (one of which is hidden at the back) are homogeneous and so can share copies of the actual data. The reference counts are indicated at the bottom of the figure. Explicitly storing block data for only four out of the eight blocks gives us a memory saving of 50% in this overly simplistic example.

Compression in this system arises because, if two blocks have identical contents, then we are able to have both entries in the block list pointing to the same block data. This occurs frequently with blocks that are completely homogeneous. It is of course also possible that two heterogeneous blocks also happen to be identical, but this is sufficiently rare that it is not worth the computational overhead involved in checking for it.

An obvious question is why we bother to store a homogeneous block at all, rather than simply setting a flag to indicate that it is homogeneous and then storing its value.

This is a perfectly valid approach and does save a little more memory, but it means that each time a voxel is accessed, we must add some additional logic to determine whether we should follow a pointer to some block data or just use the homogeneous value.

Our system adds an extra layer of indirection when accessing voxels because we must first determine in which block a voxel is located, follow the pointer, and then access the voxel. However, we have also already emphasized the importance of having fast access to the neighboring voxels, and these are also likely to lie within the same block as each other. Hence, we have found it useful to introduce the concept of a *volume sampler* object which, once pointed at a voxel, will cache the block lookup in order to speed up access to other voxels in its locality.

When writing to voxels there are a couple of scenarios for which we need to watch out. First, the block to which we are writing might currently be shared. We must check the reference count of the block and duplicate the data if necessary. Second, the act of modifying the data may cause the block to become homogeneous and therefore eligible for sharing. This is costly to verify as it potentially involves reading and comparing every voxel in the block. So instead we mark the block as being *potentially homogeneous* and provide a garbage collection routine that can be called whenever there is spare processing time (e.g., the CPU is stalled waiting on some other task).

Lastly, we need to take some care to choose an appropriate block size and there are several factors that can influence this:

- Smaller blocks have a greater chance of being homogeneous, and therefore of being shared. On the other hand, this means there will be a greater number of blocks and so the block list will be longer. The effect this has on memory usage can be seen in Table 3.1.
- Larger blocks have a smaller proportion of voxels on the faces of the block. This means the volume sampler is more effective because it is less likely to need to look at neighboring blocks during voxel neighborhood operations.

- Blocks are required to have a side length that is a power of two, in order to allow addressing operations to be implemented using bit manipulation.
- Blocks should be small enough to fit into the CPU cache.

Table 3.1: The memory required to store our example volumes varies depending on the block size.

Volume	Dimensions	Uncomp.memory	Block size	No. of blocks	Comp.memory	Comp.ratio
Castle	256×256×256	16 MB	16×16×16	4096	1.66 MB	89.6%
	256×256×256	16 MB	32×32×32	512	2.87 MB	82.1%
	256×256×256	16 MB	64×64×64	64	4.51 MB	71.9%
Mountain	512×512×256	64 MB	16×16×16	16384	12.98 MB	79.8%
	512×512×256	64 MB	32×32×32	2048	21.95 MB	65.8%
	512×512×256	64 MB	64×64×64	265	34.51 MB	46.1%
Earth	512×512×512	128 MB	16×16×16	32768	19.27 MB	84.9%
	512×512×512	128 MB	32×32×32	4096	39.34 MB	69.3%
	512×512×512	128 MB	64×64×64	512	68.51 MB	46.5%

Given the above constraints, Grimm et al. found that a block size of 32×32×32 gave the best overall performance. Our experimental results in Table 3.1 show that we can expect about a 70% compression rate in this scenario.

^[2]The current version of our engine does not use a density value (only a material ID) for each voxel. This is because we have been focused upon representing other structure beyond terrain. However, there is a fork of our codebase [5] that has had this

functionality added.

3.4 Surface Extraction

We have already introduced the principles of the Marching Cubes algorithm (and we suggest the reader consult [7] for a more detailed explanation). In this section we examine some of the practicalities of implementing the algorithm with respect to our specific application. These include:

- Allowing the data to be modified in real time, and intelligently regenerating the surface for the modified part without running the algorithm over the entire volume again.
- Outputting the resulting surface mesh in a format that is suitable for GPU rendering or for further processing.

To aid with the above, we break our volume down into adjacent and non-overlapping cubic *regions*. This means that every cell is in only one region, but voxels on the face, edge, or corner of a region also belong to neighboring regions. Each of these regions is assigned separate vertex and index buffers to hold the mesh data corresponding to the surface that is contained in that region. Additionally, each region keeps a dirty flag that indicates whether the triangle data in the buffers matches the current volume data for that region.

Although this may sound similar to the breaking down into blocks discussed earlier, it is important to realize that the two concepts of *regions* and *blocks* serve entirely different purposes. *Blocks* are a data representation used to provide efficient storage and fast access to the voxels, where as *regions* are used to restrict the execution of the Marching Cubes algorithm to the parts of the volume that have changed, and also for the purpose of visibility culling.

After the volume is set to its initial value, the Marching Cubes algorithm is executed to update the mesh data for each region, and the dirty flag is cleared. Further attempts to write to the volume result in the regions containing the affected voxels being marked as dirty. A trivial check is performed to ensure that voxels are not being set to their current value. Also, remember that most voxels only belong to a single region, but those on the face, edge, or corner of a region are shared by neighboring regions as well. Hence, modifying a single voxel can cause multiple dirty flags to be set.

Any dirty region will need to have its mesh regenerated, but this should not happen immediately because it is an expensive operation, and it is likely that other nearby voxels (probably in the same region) are also about to be updated as a result of the user's current action. Regeneration should instead be delayed until all volume modifications for the current frame are complete.

As with the block size, there are a number of factors to take into account when choosing an appropriate region size:

- Smaller regions mean more regions, resulting in a higher batch count. For each region that contains a surface, we generate at least one batch (possibly more than one batch if we split materials as described in Section 3.5.2). Batch count is one of the limiting factors in the performance of modern GPUs.
- Smaller regions offer the opportunity for finer-grained occlusion culling (see Section 3.5).
- Smaller regions will typically enclose a modified part of the volume more tightly, meaning fewer cells have to be processed by the Marching Cubes algorithm.
- Large regions take longer to regenerate.
- Large regions may contain more than 65,536 vertices, which will then require 32-bit indices. This has some memory overhead, and may not be supported on older hardware.

- Matching the region size to the block size makes it easier to implement the Marching Cubes algorithm in a cache-friendly manner.

As a guideline, we break our volume down into $8 \times 8 \times 8 = 512$ regions.

In addition to generating the vertex positions, we also require vertex normals to perform shading calculations. There are several known approaches [17] to computing these normals from the mesh data but these can suffer from mismatches on the region boundaries. An alternative is to generate the normals directly from the volume data by computing the gradient vector.

An approximation of the gradient vector can be found using the *central difference* operator [12]. For a voxel at integer position (x, y, z) , the central difference gradient can be found by examining the density value of neighboring voxels as follows:

$$\nabla f(x, y, z) = \frac{1}{2} \begin{bmatrix} f(x+1, y, z) - f(x-1, y, z) \\ f(x, y+1, z) - f(x, y-1, z) \\ f(x, y, z+1) - f(x, y, z-1) \end{bmatrix}$$

If a smoother gradient is required then the *Sobel* operator [12] may be used instead, but in general the central difference operator will be sufficient for our purpose. The gradient at an arbitrary point in the volume can be found by interpolating the gradients from the corners of the corresponding cell. When finding the gradient at a given vertex position, a one-dimensional interpolation is sufficient because a generated vertex will always lie on a cell edge.

Output of the Algorithm

For each region, the algorithm generates a single vertex and index buffer pair. In the case that the region does not contain a surface, the buffers will be empty and need not be uploaded to the GPU. As mentioned, the indices may be either 16 or 32 bits depending on the number of vertices, and the vertices contain the following information:

```
struct Vertex
{
    float position[3];
    float normal[3];
    float materialId;
    float alpha;
}
```

The material ID is stored as a float for compatibility with Shader Model 3.0 hardware. If you are targeting a more modern GPU, then you may want to use an integral type for direct use as an index into a texture array (see Section 3.5.1). The alpha value is used to blend between different materials and will be discussed further in Section 3.5.2.

Although we have not yet implemented it, one useful optimization when rendering geometry that is represented by buffers is to adjust the order in which primitives are rendered in order to effectively utilize the GPU vertex cache. Primitives that are close together spatially should also be close together in the buffer, such that when rendering there is an increased likelihood that a cached vertex can be reused rather than the vertex shader being executed.

Both Nvidia and ATI provide offline tools for this job but we require something that runs quickly on meshes generated at run time. There are some possible candidates for this [16], but it remains to be seen if their performance is sufficient for our application.

Level of Detail

Another important technique for improving the rendering performance of our system is to implement some kind of level-of-detail (LOD) mechanism such that regions that are distant from the camera are represented with fewer triangles than those that are close to it.

Within our engine, we implemented LOD directly on the volume, rather than on the generated triangle surfaces. That is, we build a mip pyramid where each level has half the width, height, and depth of the level below it. (This is conceptually similar to texture mipmaps used on graphics cards.) We keep the same number of regions such that each region also has half the dimensions of its predecessor in the pyramid.

We then run the surface extraction on the appropriate mip level based on the distance of the region from the camera. As the camera moves around regions can have their surface regenerated at a different resolution, with the previous surface either being discarded or cached for possible later use. The threading system that handles this automatic regeneration in the background is discussed in Section 3.4.3.

To actually generate the mip pyramid it is necessary to be able to derive the value of a voxel from the eight voxels in the preceding mip level. If we are storing density components for our voxels then these can simply be averaged. If we are storing a material ID then it does not make sense to average these (the combination of two different materials should be one of those two, rather than a third material). In our system we take the minimum, although the most frequently occurring value (the statistical mode) could also be chosen.

One drawback of our LOD mechanism is that there tend to be rather large cracks between adjacent regions using different LOD levels. This is a classic problem in terrain rendering and has been the focus of significant research for the 2D height map scenario, but it is vastly more difficult in three dimensions. Lengyel [6] has solved this problem using a more sophisticated set of equivalence classes that consider both the higher and lower resolution data when generating triangles.

In addition to the discrete LOD system outlined above, we also have an early version of a *progressive* LOD system based on the work of [11] and [18]. It is little more than a vanilla implementation of the techniques described in these papers, with the

exception that we modify the edge collapse heuristic to not collapse edges that lie on material boundaries or on the boundaries of regions. This eliminates the cracks between regions but the system is significantly slower than the discrete LOD approach. In particular, generating a low level-of-detail mesh with the progressive system takes at least as long as generating a high-resolution mesh, plus the time taken to perform the simplification. In contrast, our current discrete approach can generate a low LOD in much less time than the higher LOD.

Threading the Algorithm

Interactive volume editing is crucial for allowing designers to build their 3D worlds, and within our engine we like to think of it as a game play feature as well. Therefore, it is essential that the regeneration of a dirty region's surface is performed as quickly as possible. Efficiently threading the algorithm can go a long way in helping us achieve this goal.

Our system is illustrated in Figure 3.5 and works as follows. Our main game thread handles logic and rendering and is the only one to have both read and write access to the volume. Voxels are modified based on user input and in-game events, and the region's dirty flag is set. We periodically collect dirty regions and populate a simple `TaskData` structure with the region, the required LOD level, and a priority. This priority is based upon the region's distance from the camera, such that nearby regions will be regenerated sooner. The task data is added to a prioritized queue of tasks that need to be processed. If the task is already in the queue then there is no need to add it again, as the corresponding region is already scheduled for an update.

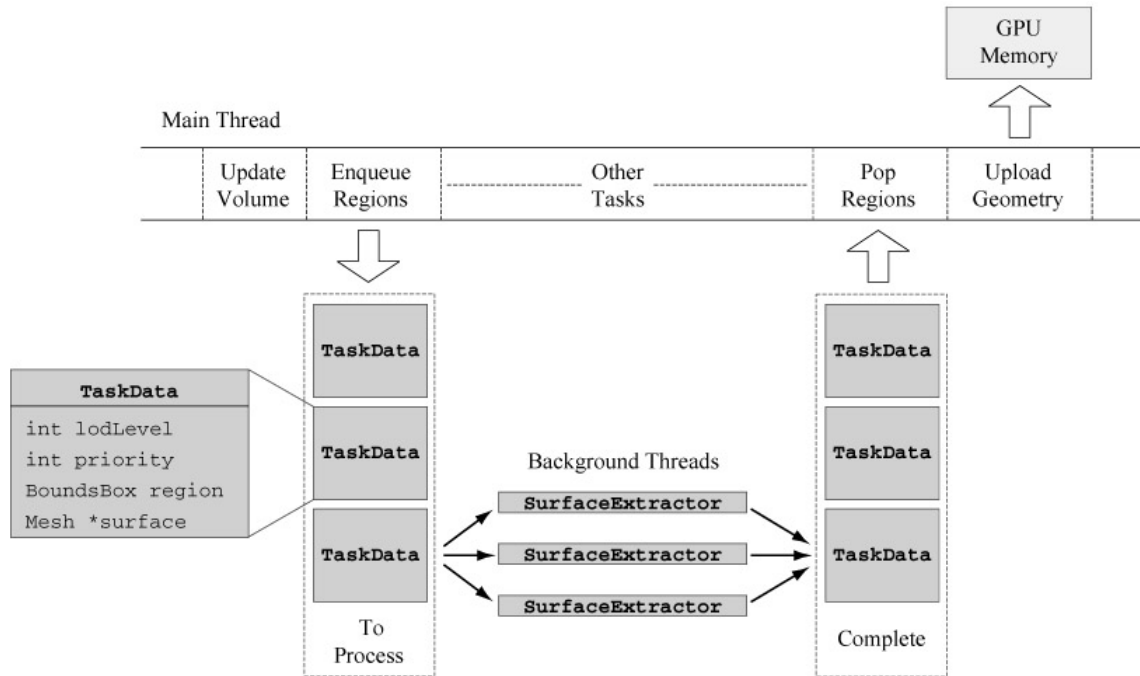


Figure 3.5: A small number of background threads continuously process the queue of regions that have been modified and regenerate the surface geometry. The main thread retrieves the results and uploads the geometry to the GPU.

A small number (1–4) of surface extraction threads run in the background and wait for the priority queue to contain task data to process. When a `TaskData` instance is available, a surface extraction thread will remove it from the priority queue and perform the Marching Cubes algorithm on its specified region. If other mesh processing tasks such as normal computation (Section 3.4) or splitting by material (Section 3.5.2) are necessary, then they are also performed by the surface extraction thread. The resulting vertex and index buffer pair is then assigned to the `TaskData` instance, which in turn is added to a queue of completed tasks.

The main thread is the only one that can copy the data to the GPU (at least on

current generation hardware), and so this takes responsibility for removing items from the queue of completed tasks and uploading them to the graphics card. Semaphores are used to control access to the queues and to provide thread synchronization.

In order to give an indication of how well our threaded surface extractor performs, we ran each of our three test volumes through the surface extractor using between one and four threads. In each case, we measured the amount of time actually spent performing surface extraction. Time spent performing other tasks such as loading the volume from disk and uploading the meshes into GPU memory is omitted from these results, which can be seen in Table 3.2.

Table 3.2: Some typical timings for our threaded surface extractor running on a quad-core 2.33 GHz CPU with 2 GB of memory.

	No. of threads	Thread 1 time (ms)	Thread 2 time (ms)	Thread 3 time (ms)	Thread 4 time (ms)	Average time (ms)	Sum of times (ms)	No. of regions	Time per region (ms)
Castle	1	1094	-	-	-	1094	1094	64	17
	2	501	589	-	-	545	1090	64	17
	3	360	439	344	-	381	1143	64	18
	4	281	280	265	322	287	1148	64	18
Mountain	1	5292	-	-	-	5292	5292	256	21
	2	2717	2683	-	-	2700	5400	256	21
	3	1960	1888	1955	-	1912	5736	256	22
	4	1438	1340	1314	1518	1402	5610	256	22
Earth	1	9062	-	-	-	9062	9062	512	18
	2	4439	4455	-	-	4447	8894	512	17

Table 3.2: Some typical timings for our threaded surface extractor running on a quad-core 2.33 GHz CPU with 2 GB of memory.

	No. of threads	Thread 1 time (ms)	Thread 2 time (ms)	Thread 3 time (ms)	Thread 4 time (ms)	Average time (ms)	Sum of times (ms)	No. of regions	Time per region (ms)
	3	2926	3063	3045	-	3011	9034	512	18
	4	2549	2501	2510	2537	2524	10097	512	18

There are several things we can observe from this data. First, we can see that with several threads running the workload is split fairly evenly among them. A direct result of this is that the average time spent in a thread decreases as the number of threads increases. It's not quite a linear relationship, but it is pretty close, as shown by the "Average time" column. Second, we can see that the time to regenerate a region ($64 \times 64 \times 64$ in all these cases) is consistently around 20 ms. This means we would typically expect to pick up the results of modification 1–2 frames after it has occurred, and it makes $64 \times 64 \times 64$ the largest region size we can practically use for real time modification. Lastly, we see that the time required to process a complete volume is typically just a few seconds.

3.5 Rendering

For the purpose of rendering, a bounding volume hierarchy is built with our region's geometry as the leaves. The bounding box of each piece of leaf geometry is at most the size of a region, but is usually smaller and it takes relatively little effort to trim the bounding box to the size of the actual mesh. Each internal node of the hierarchy is built from the eight nodes below it.

Because of the highly dynamic nature of our environments we do not perform any

kind of precomputed visibility calculations. Visibility culling is currently handled simply by intersecting the view frustum with the bounding volume hierarchy, and rendering each piece of geometry whose bounding box is at least partly inside. This is a fast and efficient method for culling large amounts of geometry, but it is not an optimal solution for scenes with a high depth complexity. For these we are investigating the use of image-space methods such as Coherent Hierarchical Culling [4], but we do not yet know how well these will perform.

Lighting and shadowing algorithms are also fully dynamic, as the ability to modify the volume at any time makes the use of precomputed illumination very difficult. Normals for lighting are usually provided with the vertices (as discussed in Section 3.4.1), but in some cases it is possible to generate them on the fly. So far, we have applied only local illumination models, but some of the current research on real time global illumination may also be applicable. One of the many variants of the popular shadow mapping algorithm can be used to generate the real-time and dynamic shadows.

The Material System

When a surface is rendered, it is almost always desirable to provide additional surface detail to the object through the use of texture mapping. One of the key problems with generating geometry on the fly is that there is no opportunity for an artist to define the UV parameterization that specifies how these texture maps should be applied. Instead we need an automatic way of generating texture coordinates.

One of the most useful approaches (demonstrated in real time by Nvidia in their Cascades demo [3]) is known as *triplanar texturing*. This is performed in the fragment shader and uses blend weights derived from the surface normal to interpolate between three textures projected along the x , y , and z axes. That is, the first texture is sampled using the (y, z) components of the fragment's world-space position and modulated by the x component of the normal, the second texture is sampled using the (x, z)

components of the fragment's world-space position and modulated by the y component of the normal, and so forth. A fragment's world-space position can be determined by interpolating it from the vertices, and the components of a normal vector can be made to sum to one by squaring them. The snippet of Cg fragment shader code in Listing 3.1 demonstrates this process.

Listing 3.1: Triplanar texturing can project textures onto arbitrary geometry. This code receives textures, a normal, and a world position as input and computes the resulting color. Note that in order to preserve texture handedness, one of the UV coordinates must be negated whenever the dominant normal component is $-x$, $+y$, or $-z$. This is particularly important when working with normal maps but is not shown for simplicity.

```
// Interpolation means normals may not be unit length
normal = normalize(normal);

// Squaring a unit vector makes the components add to one.
float3 blendWeights = abs(normal * normal);

// For each axis, sample the texture and multiply by the blend weights.
float4 colorMapValueYZ = tex2D(colorMapX, worldPos.yz) * blendWeights.x;
float4 colorMapValueXZ = tex2D(colorMapY, worldPos.xz) * blendWeights.y;
float4 colorMapValueXY = tex2D(colorMapZ, worldPos.xy) * blendWeights.z;

// Combine the results
float4 colorMapValue = colorMapValueXY + colorMapValueYZ +
    colorMapValueXZ;
...
```

Triplanar texturing works particularly well when applied to terrain, and when natural textures (rock, grass, etc.) are used. It does not respond particularly well to manmade textures containing sharp edges or other high frequency detail, as these do

not blend well with each other. Variations on this idea are also possible, such as using six textures instead of three, or using one texture for all four lateral surfaces and different textures for the top and bottom.

An alternative method is to use the normal directly to lookup into a cube map texture. This is what was done for the surface of the Earth in Figure 3.1(b). In this case, the normals which were used for the lookup were generated on the fly by normalizing a vector from the center of the Earth to the vertex on the surface. This gave better results than using the normals generated by the methods in Section 3.4.

However, the conceptually simplest way to perform the parameterization is to use three-dimensional texture coordinates. In this case, the texture coordinates can be derived directly from the fragment's world-space position. Storing 3D textures on the GPU quickly becomes impractical because of their high memory requirements, but the 3D texture coordinates can easily be used as inputs into *procedural* texture generation routines. Referring again to Figure 3.1(b), the lava in the Earth's core is generated by using several octaves of Perlin noise running on the GPU [10].

Note that whichever technique is used to generate the texture coordinates, it is still possible to perform texture transformations by applying an appropriate texture matrix.

Using Multiple Materials

If our voxels include a material ID, then this will have been passed to the GPU as part of our vertex definition (see Section 3.4.1). One simple way in which we can use this material ID is to identify the texture (or set of textures for triplanar texturing) that should be applied. Modern GPUs provide direct support for this through *texture arrays*, which allow a single 2D texture to be indexed in an array of textures using the rounded value of a floating-point input. Older hardware can make use of a *texture atlas* [14] to obtain a similar result, but special measures must be taken to avoid filtering artifacts when textures repeat.

Although simple in principle, there are some additional issues to be wary of if you wish to blend smoothly from one material to another. For example, a triangle on the boundary of two or more materials will have a different material ID at each of its vertices. It does not make sense to simply interpolate these material IDs across the face of the triangle, as this would yield values which did not exist at any of the three vertices.

Within our system we handle this scenario by splitting our input mesh in two. One of the resulting meshes contains only those triangles that have the same material at each vertex (we will call these *uniform* triangles). The other *nonuniform* triangles are replaced with new triangles that have a material ID of zero at each vertex. During shading, we ensure that material zero is drawn as black by either setting the zeroth slot of our texture atlas to black, or by putting in an explicit check and return at the beginning of our fragment program.

The second mesh that results from our splitting procedure contains the non-uniform triangles. Actually, each non-uniform triangle gets duplicated three times^[3] to create three uniform triangles, one for each of the materials in the non-uniform version (see Figure 3.6). We set the alpha values of the vertices such that one corner is fully opaque while the other two are fully transparent.

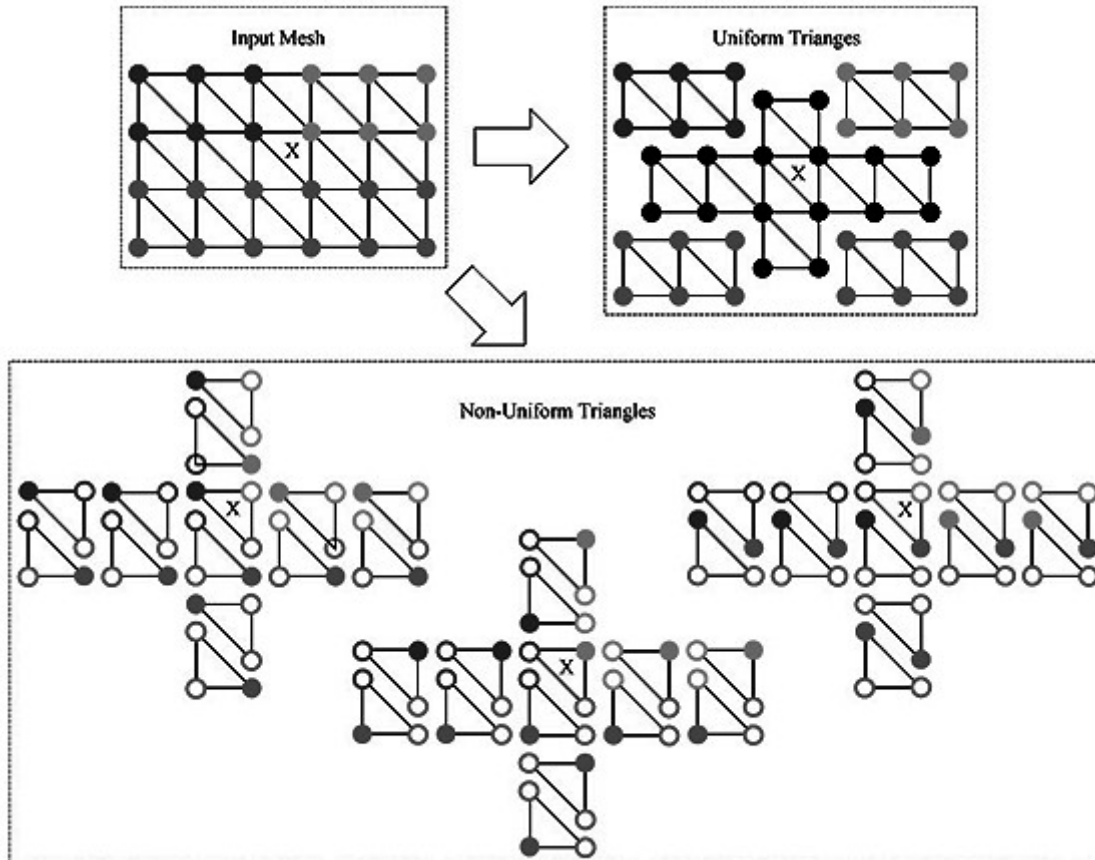


Figure 3.6: A single input mesh containing multiple material IDs, represented by different colors (see figure on accompanying CD), is split into two meshes. In the uniform triangle mesh, all the components are spatially adjacent and are only split up in the figure to aid visualization. In the nonuniform triangle mesh, the three parts are drawn on top of each other such that the alpha values blend correctly.

Rendering is performed by first drawing the uniform triangles with the blending mode set to replace the current contents of the frame buffer. As well as drawing the uniform triangles, this also serves to ensure that the background behind the non-uniform triangles is set to black. The blending mode is then set to additively blend with

the existing frame buffer contents, and the mesh containing the non-uniform triangles is drawn. This results in a smooth transition from one material to another. If we follow a single triangle such as the one marked X in the figure, we can see that it is drawn first in black and then once again using each of the materials.

Note that this triangle duplication on material boundaries is currently performed on the CPU after the meshes have been generated by the Marching Cubes algorithm. This is done for compatibility with older hardware, but it would be interesting to investigate whether it could instead be handled by the geometry shader on more modern GPUs.

It is possible that the material represented by the material IDs differ by more than just the textures that are applied. In fact, the entire shaders and/or pipeline state might need to be different for some materials. Figure 3.1(b) is again a good example of this, as it uses a cube map projection for the surface of the Earth, triplanar texturing for the rock, and GPU Perlin noise for the magma.

If the mesh corresponding to a single region does need to cater for such a diverse range of materials, then we split the mesh into several pieces, up to the number of different materials. For example, if a mesh consists of three different materials, of which two are based on triplanar texturing and the third is procedurally generated, then we split the vertices for the third material, but leave the first two materials in the same mesh. We then render the first mesh using our triplanar texturing shader (choosing among textures based on the material ID) and then render the second mesh using our procedural shader. This splitting into materials is *in addition* to the splitting described earlier for blending among materials. This means that if a region contains n different materials, then it may end up in at most $2n$ pieces after all splitting is complete. However, this is a worst case scenario, and in practice many region's meshes do not need to be split at all.

Our system benefits here from being built on the Ogre3D graphics engine [9], as we can simply pass our meshes into Ogre's render queue and the sorting by texture changes, render state changes, etc., is handled automatically. Most graphics engines will provide some similar functionality.

^[3]Actually there is some room for improvement here as triangles containing two materials should only be duplicated twice, rather than duplicating all non-uniform triangles three times. We intend to change this as triangles consisting of three different materials are quite rare, and so a lot of extra triangles are currently generated to support this worst-case scenario.

3.6 Physics

Integration of a physics solution (in our case Bullet [2]) into our engine was relatively straightforward, as for the most part, the geometry generated by the Marching Cubes algorithm can be treated the same as any other. A physics mesh is constructed for each region from the vertex and index buffers, and the same bounding volume hierarchy that we used for view frustum culling can be used for the broad-phase collision detection.

During simulation, we found that the sheer number of triangles did put the physics system under a lot of strain, and that an effective LOD system becomes essential for volumes with dimensions over about 256^3 (of course, this varies wildly depending on the complexity of the volume). Our existing LOD system is not particularly suitable for this purpose due to the mismatches between LOD level alignment discussed in Section 3.4.2. Simplifying the original high-resolution meshes would likely be an improvement here.

Dynamically updating the meshes as voxels are removed was a lot more straightforward than initially anticipated. Simply replacing one mesh with another

between simulation time steps seemed to cause the physics engine no problems at all. Dynamically adding voxels is a lot more complex because an object can suddenly find itself penetrating a surface that it was previously not close to. More work is required to decide how or if this scenario should best be handled.

One additional point worth noting is that while all collision detection is currently performed against the surface meshes, there is potential for doing it directly against the voxel volume. Hit testing and picking is currently performed in this way, and testing whether a point is inside an object or not becomes a simple case of checking the value of the voxel at that location.

3.7 The Future

Most of the techniques described in this chapter have been implemented in our experimental Thermite3D game engine [15], with the exception of the "density voxels" which are available in the *Forever War* spin-off project [5]. The techniques are appropriate for integration into other game engines running on current generation hardware. However, our project is currently at an early stage, and there are a number of features we would like to add in the near future.

First, we would like to increase the size of the volumes that we can load and render. This will require further work on our LOD system, particularly with the aim of determining how the progressive LOD approach compares to the current discrete system. Second, we would like to investigate the use of streaming as a mechanism for reducing the amount of data held in memory at a time. Our current block volume structure is likely to provide a strong basis for this as blocks can be stored in a compressed format on disk, and loaded into memory on demand. We may also want to save blocks back to disk so that changes made to the environment can be persistent.

In addition, we need to give some thought to the issue of *content creation*. At present we have tools to convert existing height maps into volumetric representations so they can be destroyed in real time, and we also have a tool that will convert a triangle mesh from a standard 3D modeling package into a volume (this was used to create the castle in Figure 3.1(c), for example). But there is a lot of potential for generating environments procedurally, such as a terrain built from Perlin noise with an underground network of caves built using Voronoi cells.

It is also important to consider how the use of volumetric environments can be used as a game play element. The ability to destroy parts of the environment in response to explosions is an obvious example, but the representation also lends itself naturally to allowing parts of the environment to be slowly eroded away, perhaps in response to fire or acid. If a game scenario required it, it would also be possible to slowly heal geometry back to its initial state. Lastly, the development of a powerful and intuitive interface for editing could also help its adoption as a game-play device.

Finally, if we look beyond our own project, it is worth noting a significant amount of research on other methods of rendering these kinds of environments. id software is probably the most high profile of these with talk of a "Sparse Voxel Octree" being used to represent geometry in their id Tech 6 engine [13]. As GPUs become increasingly general purpose, it is becoming practical to implement other volume rendering approaches such as ray casting or point splatting. For now, the surface extraction approach described in this chapter is the only approach to have seen use in real games, but it will be interesting to see where the future leads.

Acknowledgements

I would like to thank Matthew Williams and Jaz Wilson for their contributions to the Thermite 3D Engine, and the developers of Ogre3D and Bullet for their valuable libraries. Thomas Schöps and Marek Rosa granted permission to use their images in

this gem, while Tobias Tropper provided useful feedback on early versions of the gem.

References

- [1] Sören Grimm, Stefan Bruckner, Armin Kanitsar, and Meister Eduard Gröller. "A Refined Data Addressing and Processing Scheme to Accelerate Volume Raycasting". *Computers & Graphics*, Volume 28, Number 5 (October 2004), pp. 719–729.
<http://www.cg.tuwien.ac.at/research/publications/2004/grimm-2004-arefined/>
- [2] Erwin Coumans. Bullet Physics Library. <http://bulletphysics.com/>
- [3] Ryan Geiss and Michael Thompson. "NVIDIA Demo Team Secrets—Cascades", Game Developers Conference 2007.
<http://developer.download.nvidia.com/presentations/2007/gdc/CascadesDemoSecrets.zip>
- [4] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. "CHC++: Coherent Hierarchical Culling Revisited". *Computer Graphics Forum (Proceedings Eurographics 2008)*, Volume 27, Number 2 (April 2008), pp. 221–230.
<http://www.cg.tuwien.ac.at/research/publications/2008/MATTAUSCH-2008-CHC/>
- [5] Thomas Schöps and Oliver Schneider. *Forever War*. <http://foreverwar.sourceforge.net/>
- [6] Eric Lengyel. "Voxel-Based Terrain for Real-Time Virtual Simulations". Ph.D. diss., University of California, Davis, 2010.
- [7] William E. Lorensen and Harvey E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm". *ACM SIGGRAPH Computer Graphics*, Volume 21, Number 4 (July 1987).
- [8] Keen Software House. *Miner Wars*. <http://www.minerwars.com>
- [9] Steve Streeting. *Object-Oriented Graphics Rendering Engine*. <http://www.ogre3d.org>

- [10] Simon Green. "Implementing Improved Perlin Noise". *GPU Gems 2*, Addison-Wesley, 2005. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter26.html
- [11] Stan Melax. "A Simple, Fast, and Effective Polygon Reduction Algorithm". *Game Developer Magazine*, November 1998. <http://www.melax.com/polychop>
- [12] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Christof Rezk-Salama, and Daniel Weiskopf. *Real-Time Volume Graphics*. AK Peters, 2006. <http://www.real-time-volume-graphics.org/>
- [13] Jon Olick. "Current Generation Parallelism In Games". *Beyond Programmable Shading*, Siggraph 2008. <http://s08.idav.ucdavis.edu/olick-current-and-next-generation-parallelism-in-games.pdf>
- [14] Nvidia. "Improve Batching Using Texture Atlases". July 2004. http://http.download.nvidia.com/developer/NVTextureSuite/Atlas_Tools/Texture_Atlas_Whitepaper.pdf
- [15] David Williams. *Thermite3D Game Engine*. <http://www.thermite3d.org/>
- [16] Gang Lin, and Thomas P.-Y. Yu, "An Improved Vertex Caching Scheme for 3D Mesh Rendering". *IEEE Transactions on Visualization and Computer Graphics*, Volume 12, Number 4 (July 2006). <http://www.ecse.rpi.edu/~lin/K-Cache-Reorder/>
- [17] Shuangshuang Jin, Robert R. Lewis, and David West. "A comparison of algorithms for vertex normal computation". *The Visual Computer*, Volume 21, Numbers 1–2 (February 2005), pp. 71–82. http://www.tricity.wsu.edu/cs/boblewis/pdfs/2003_vertnorm_tvc.pdf
- [18] Tom Forsyth. "Comparison of VIPM Methods". *Game Programming Gems 2*, Charles River Media, 2001. http://home.comcast.net/~tom_forsyth/papers/gem_vipm_webversion.html
- [19] "Virtual Terrain Project". <http://www.vterrain.org/>

4

High-Level Pathfinding

Daniel Higgin

Lunchtime Studios, LLC

Overview

Today's gamers demand spectacular pathfinding. If it isn't amazing, don't ship it. Otherwise, prepare for an assault of angry gamers mobbing the developer's studios, torches ablaze and pitchforks in hand. Simply put, pathfinding is one of the most important pieces of technology to get right in a game, especially in the real-time strategy genre. Besides the pressure of perfection, pathfinding programmers need a thick skin. These brave souls should assume that gamers don't compliment path-finding, they roast it. Gamers latch on to almost any game or AI imperfection and blame it on the pathfinding system. In truth, they used to be right. The old generation of pathfinding engines resulted in actors looking like knuckleheads as they found less than optimal paths and often failed to find a valid path altogether when one existed.

Apart from the impact pathfinding has on gameplay, it also has an enormous effect on performance. Game lag should never be the result of pathfinding, but sadly, it happens all too often. What gamers don't realize is that behind the scenes, poor pathfinding results in a restricted game world. These restrictions keep gamers from experiencing a game designer's true vision, which is never a good thing. Certainly not all genres need fast pathfinding, but for those with large worlds or many actors,

pathfinding must be accurate and optimized. Like many of the cleverly designed usability features of modern software, pathfinding that goes unnoticed by the user is a success.

There are many options for optimizing pathfinding. We can optimize our code, time-slice pathfinding calculations, reuse paths, group paths, and even avoid pathing all together when we can get away with it. One optimization however, really stands out, and that is to reduce the search space a pathfinder works with. High-level pathfinding is a great technique to achieve this reduced search space, and is the focus of this gem. Besides dramatically improving performance, high-level pathfinding offers us the opportunity to perfect pathfinding. Once pathfinding performance is no longer an issue, we can force our pathfinding engines to never give up, and always find a path when a valid one exists.

4.1 Terms

Before we dive into how high-level pathfinding works, let's review some terms.

- **Actor.** Any significant object within a game's world that serves a purpose beyond ambience.
- **RTS.** Short for "real-time strategy", a genre of strategy games that focuses on unit production, resource management, and for some, combat. Its typically high actor count and large world area strain traditional pathing algorithms, which are designed for smaller datasets.
- **Tile.** A square area of a world (such as 1×1 meters) that holds properties for that given world space. In a typical RTS game, tiles are laid out in a grid pattern and can be located with integer coordinates.
- **Path.** A sequence of points through which an actor can legally move to get from one location to another.

- **Detailed path.** Exact tiles an actor will navigate to get to their destination. This path avoids all obstacles.
- **Path rule.** A game rule that dictates a movement restriction for a given actor type, such as whales not being allowed on land. These rules determine what tile types are "legal" for an actor's movement.
- **Path region.** A collection of contiguous tiles that share the same path rules.
- **Beacon point.** The path region's point closest to the region's center of mass.
- **Fuzzy pathing.** Fuzzy pathing is another way of saying high-level pathing. It indicates that a non-detailed path is being created that is legal, but without many of the movement details that make paths look good and avoid all obstacles. Another way to think of a fuzzy path is that it indicates there is a guaranteed way to the destination via this route, but it may have some unknown and navigable obstacles along the way.
- **Terrain analysis.** A term for computing knowledge of a game world [3]. It's data we compute about a world and organize in a way so that, from within a game, the pathfinder can approximate a human-like awareness of the environment. Examples include computing ocean tiles and shapes, recognition of a bay, a river, marshlands, forests, beaches, etc.

4.2 Start Your Engines

For the best understanding of this article, knowledge of pathfinding isn't essential, but is strongly recommended. What is contained within this article is a skeleton view of what's required in a high-level pathfinding system. While reading, it's critical to remember that even with brilliant designers and genius programmers, there will be frequent revisions to the path rules to handle special cases until pathfinding is perfect and feature freeze occurs.

Truly, the best way to build a high-level pathfinding system is to make it as generic

as possible with known and isolated customization points. This is a perfect time to either buy, or dust off those design pattern books. Be ready to utilize patterns such as factories, strategies, prototypes, and policies. Isolate the customization points, because those will be frequently modified, and we don't want tons of special case code floating around the engine.

4.3 Why High-Level Pathfinding?

Imagine we have a world that contains over a million tiles. In this world, there is a battle raging in the North at the walls of an enemy city as seen in Figure 4.1. Our heroic army is trying to take the walled in city by force, but needs reinforcements. Let's help our army by moving a siege ram, a man-at-arms, and naval galley north to the battle.

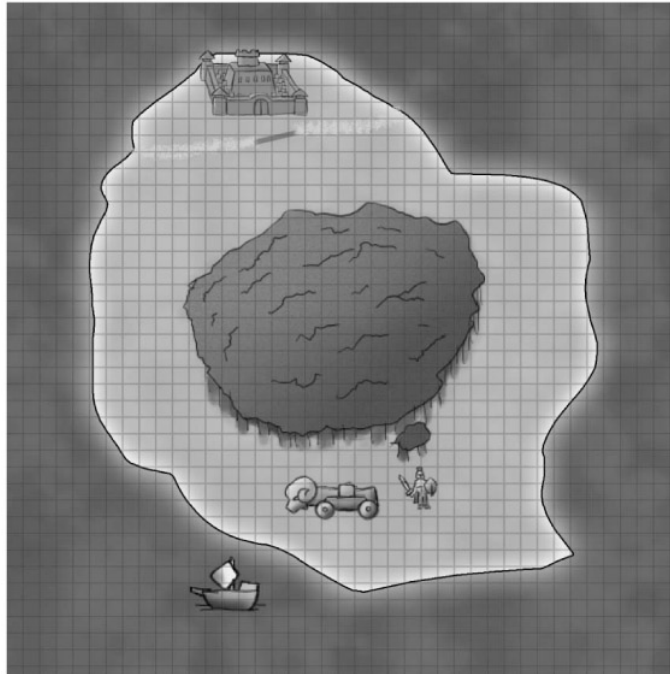


Figure 4.1: A world map, with tile markings.

A million or more tiles is a great distance to navigate, especially with obstacles such as forests and towering enemy walls. Odds are that during a battle, CPUs are busy with animations, projectiles, physics, AI, and graphics. The effort of calculating detailed paths for an army across a huge map will adversely affect game performance; however, if our world was only a few thousand tiles, we could achieve this without any noticeable impact on the frame rate.

To accomplish this we need to reduce the search space to navigate our maps. High-level pathfinding achieves this through a several-phase process. First, our preprocess phase must identify each tile in the world as being of one, and only one, path type. With those types defined, we need to analyze the world and build up a knowledge base of terrain information before the game begins, which we accomplish by performing terrain analysis [3]. The second phase involves using the data stored in our world knowledge base to find a fuzzy, non-detailed path between two points using an algorithm such as A*. Our final phase involves the actor refining the fuzzy path to plan the exact route to their destination.

Sound like more than a few days of work? It should. Implementing high-level pathfinding can consume a significant amount of development time, but is well worth the effort.

4.4 Preprocess Phase

The preprocessing phase has two main parts—design and terrain analysis. In the design portion of this phase, we focus on identifying unique path tile types. This step is done without any coding, and it relies on full knowledge of actors and their movement rules. Designers should be prepared to define every actor rule such as prohibiting submarines from approaching shore lines, or declaring that only ninjas can climb walls. A rule only needs to be added only if it has a significant affect on an actor. For example,

if movement on grass and movement on sand are the same for all actors in terms of pathfinding, then there is no need to define them as different region types. If, however, sand is illegal for a unicycle, then it needs to be its own path type.

The last part of this phase is terrain analysis, and this is where the creation of path regions occur. For an in-depth explanation of terrain analysis and its many uses, it's strongly recommended that "Terrain Analysis in an RTS—The Hidden Giant" [3] is read before beginning any actual coding.

Design Time

Prior to coding, it's crucial to categorize each tile into a unique pathfinding type. For example, a water tile is rarely just a water tile. A given water tile may be best classified as being part of a river, bay, deep ocean, or coral reef. To identify these regions, we determine if all actors have the same restriction for traversing or avoiding the tile. If any actor is designed with a constraint that modifies their passage across the tile, we make a new path type. Perhaps it's a difference in terrain type, or maybe there's a special rule for large actors. When this occurs, we may turn the special case rule into a unique type.

Let's see a few examples. In our effort to reinforce the army to the North, each of our soldiers has very different pathfinding restrictions. Our rules are as follows:

- **Siege Ram.** Big and bulky, this actor must move on roads or grasslands. What makes this actor unique is that while pathfinding, it can move through enemy walls and gates if it doesn't mind attacking them first. Siege rams also cannot cross rivers without using a bridge.
- **Man-At-Arms.** Our plated soldier can be in group formations or act independently as a scout. Unbound by the restrictions of using a horse, when alone, our knight can move through the woods if needed.

- **Galley.** This warship is large, and used primarily in ship to ship combat or to bombard the coast from a distance. River warfare is out of the question. Even approaching near to the coast could risk it running aground and is forbidden. As a result, it must stay in deep water at all times.

Already, we can see a need for different types of water and recognize that we need to separate walls, gates, forests, and bridges into separate types. It's essential to go through each type of actor that can pathfind and determine its movement rules. It's a good idea to establish an enumeration of path tile types and a function for computing them, as shown in Listing 4.1.

Listing 4.1: This is a function used to determine the path type of a tile. The order of rules here is very important. This function is a special case and will be frequently modified during development.

```
namespace PathTileType
{
    typedef unsigned long Type;
    enum
    {
        Unknown = 0,
        Grass = 1,

        DeepWater = 2,
        Forest = 3,
        Wall = 4,
        Gate = 5,
        ShoreLine = 6,
        Shallows = 7,
        Bridge = 8
    };

    // Helper used to determine a tile's path type.
```

```
extern Type ComputeTileType(const Tile *inTile);
}

// This function takes a tile and determines its path type.
// Every tile must be determined to be of one and ONLY one type.
PathTypeType::Type PathTypeType::ComputeTileType(const Tile *inTile)
{
    // Assume asserts for tile validity, etc...
    // Is this a type of water?
    if (inTile->IsWater())
    {
        // Is there a bridge here?
        if (inTile->HasActorTypeOnTile(kActorTypeBridge))
            return PathTileType::Bridge;

        // Is it shallow water?
        if (inTile->GetWaterDepth() <= 1.0)
            return PathTileType::Shallows;

        // Close to shore?
        if (inTile->GetDistanceToShore() <= 2.0)
            return PathTileType::ShoreLine;

        // Must be a deep water tile.
        return PathTileType::DeepWater;
    }

    if (inTile->HasActorTypeOnTile(kActorTypeTree))

        return PathTileType::Forest;

    if (inTile->HasActorTypeOnTile(kActorTypeWall))
        return PathTileType::Wall;
```

```
if (inTile->HasActorTypeOnTile(kActorTypeGate))  
    return PathTileType::Gate;  
  
return PathTileType::Grass;  
}
```

Notice how in Listing 4.1 the order is significant. If a tile has water on it, it's more important to know if it has a bridge on it than whether it's over shallow water. If the type of water under a bridge also mattered, the type could be `BridgeShoreLine`, or `BridgeDeepWater`. It's likely there are many permutations of types, but in the end, ensure each tile belongs to only one path tile type.

It's easy to get carried away in this phase and unnecessarily define every possible variation of a similar region, so be careful. Examine the differences between the following two cases. If there are two types of bridges in the world, a draw bridge and a foot bridge, should we create unique path types for them? A draw bridge allows large ships to pass beneath them, while a foot bridge does not. An argument could be made either way, but this situation isn't so unique that it needs to be handled with a new type. It's simple enough for a ship to ask a region what type of bridge it contains.

To find an example where a unique rule is warranted, let's examine the case of ladders in the RTS game, *Rise & Fall: Civilizations at War*. Ladders are mobile, meaning they could be packed up and moved to other sections of wall. Ladders also have a strict requirement of allowing only humans. While horses could charge up ramps to fight on top of walls, ladders were strictly horse-free zones. Path regions also presented us with the opportunity to solve the issue of positioning actors directly in front of a ladder prior to climbing. We solved this by making the tile in front of the ladder's base a one tile large path region. This ensured an actor would walk to the tile in front of the ladder before climbing. Ladders, therefore, qualified as not only a unique area, but also created

the opportunity for a separate path region that we used to handle actor positioning for ladder climbing.

Terrain Analysis

Once we have our design established, we use it in the next portion of the preprocess phase, terrain analysis. Knowledge of a game world is incredibly useful to AI programmers, and terrain analysis is vital to gaining it. It is a collection of algorithms that execute before a game begins and are refined at run time in order to first analyze, then organize, game world data. Programmers then use it creatively to imply that actors have a human-like understanding of their world.

Path regions are created by iterating through each tile, categorizing, and then clumping contiguous tile types together. These clumps are then labeled by their path types and defined as a path region. It is then necessary to provide an indicator of a region's location in the world. To accomplish this, we position a beacon point at the center of mass of the path region. Remember that regions aren't necessarily nice and neat rows of tiles. A connected forest of trees could spiral out like a spider web full of sparse, but connected trees. Thus, the true center of mass for the region could be a point inside another region. In that case the beacon point is positioned by finding the closest point inside our region to that center.

These regions, or clumps of contiguous tiles, are easy for us to visualize and understand, but there is a problem: if a path region is too large, using algorithms like A* will have difficulty producing good high-level paths. Consider a path region comprising an ocean that surrounds an island. When we want to move a ship from one side of the world to the other around the island, we have to communicate to the engine that just because it's the same ocean doesn't mean it's a simple path. If the ocean was one region, there would be nothing to navigate. The low-level pathfinder would think its start and end destination was the same region, so it must be a simple path. Not so.

We would want there to be regions that took our ship around the island, regardless of it being the same ocean. The key to doing this is to divide up the regions using a world grid, which effectively normalizes the size of all regions.

Using our path region clumps, some of which are gigantic and some of which are small, we lay a grid over the world. Large and small regions alike get subdivided by this grid, making many of the regions split at least once. Choosing the size of the grid takes some experimentation, most likely, after the entire high-level path-finding system is complete. For the sake of starting somewhere, let's use a map that is 1000×1000 tiles, and a grid resolution that is 10×10 . Given a flat map of just grass, we would have many grass regions that contained 100 tiles each. That isn't an aggressive size, and it's likely a million tile map would have larger regions, but it's a starting point. Be certain to play with the resolution numbers, adjusting them up and down depending on map size, complexity, or performance. Also, it's important to store the world grid position (black lined squares in Figure 4.2) within the path region as we'll find multiple uses for it later, such as during the fuzzy pathing phase or when recomputing path areas due to world changes.

Why don't we just make the resolution such that we have huge regions? If the number is too large, as we saw from the ocean and island example, we'll miss some navigation improvements and performance gains. The key to this system is the one-two punch of first high-level (fuzzy) paths followed by detailed level paths. Since detailed paths are more computationally expensive in large search spaces, giant regions wouldn't save us as much CPU time during the detailed pathfinding phase. Conversely, if the grid forces tiny regions, then performance gains are reduced because the search space again grows. There is always a sweet spot—don't discount the importance of finding it.

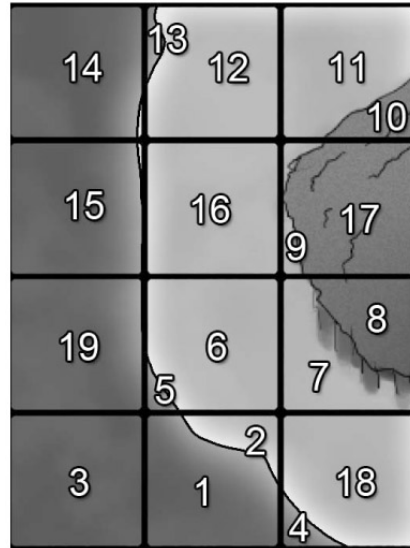


Figure 4.2: Path regions with a world grid overlay and IDs for each path region. Notice how the world grid boxes contain multiple regions inside them, and that regions like the ocean are divided up by the world grid into many separate regions.

The grid, as illustrated in Figure 4.2, shows how we can predict, within some small error amount, the distance between regions. No region spans half the map. We know the largest size, and we can make cost estimates in pathfinding based on these assumptions. If one region was gigantic and the others were small, it would be difficult to know if one was traversing a small portion of the gigantic region or traversing the longest possible distance across it. Keeping path regions sliced into a controlled grid size keeps the unknown traversal cost so low that we can predict the approximate path planning performance impact.

Adjacent Regions

In order to use all the path regions (as shown in Figure 4.2) to navigate the map, we first need to tie them together in a graph. This is done by iterating through every tile,

finding the tile's region, and finding the regions left, right, up, and down from the tile. If the region on any of the sides does not match the tile's region, we create an adjacency structure for both regions. That's fairly straightforward for left, right, up, and down, but we also need to handle diagonal connections.

Diagonals require a special adjacency rule because movement through them actually traverses neighboring tiles. Notice in Figure 4.3 that to move from region 2 to region 1, most actors would have some portion of them travel through 3 and 4 on the way. If region 3 or 4 is invalid for the actor, this diagonal must be off limits. Since we don't know what actors are going to use the connection, we need to store them somewhere and check them during high-level pathing.

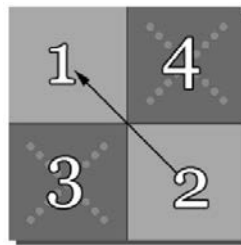


Figure 4.3: Example of diagonal connections between regions.

We don't necessarily always want to store diagonal connections. We only want to store them if the only connection between two adjacent regions is through a diagonal move. Since regions could be oddly shaped with multiple connection points to adjacent regions, there may be more than one diagonal connection point from region 2 to region 1. We need to store each instance of the diagonal connections so that pathfinding can check the boundary regions to see if the move is legal.

Do not keep any diagonal connections between two regions if a non-diagonal connection is available. Diagonal connections are only a last-case connection situation since they factor in surrounding regions during the pathing of actors. Again, even if

diagonal connections have been found between regions 2 and 1, then a non-diagonal connection between regions 2 and 1 is found, the diagonal connection information must be thrown out.

Each region should contain some structure per adjacency such as that shown in Listing 4.2.

Listing 4.2: Regions should contain a list of adjacent regions with data indicating if the connection is a diagonal, and which regions border the diagonal connection.

```
struct PathRegionConnection
{
    PathRegionConnection(PathRegionID inToRegion,
        PathRegionID inDiagonalFirst = kInvalidRegionID,
        PathRegionID inDiagonalSecond = kInvalidRegionID);

    PathRegionID mToRegion;
    PathRegionID mDiagonalA; // kInvalidRegionID if no diagonal exists
    PathRegionID mDiagonalB; // kInvalidRegionID if no diagonal exists
};
```

A World of Change

What about a dynamic world? What happens when gates fall or walls are built? We must incorporate these changes into our world's knowledge. There is no avoiding this during run time. To reduce the impact of this reanalysis, we can leverage our world grid. Given a change in the world that should affect pathfinding, we find the world grid in which the change occurred and queue it for reanalysis. Soon after, we run through all our queued world grid regions, destroy all the old path regions within these square areas, and then recompute them. When a world grid is queued for reprocessing, it's good to wait for a few moments before reprocessing since it's likely one wall built in an area will be followed soon after by another. Waiting to reprocess sections is fine when buildings

are constructed, but not when buildings are destroyed. Imagine a ram knocks down a wall, a king then issues a charge into the city, and the men begin walking in the wrong direction because they don't realize the wall no longer blocks their entry. Not good!

With all our path regions subdivided using a grid, as shown in Figure 4.2, and all adjacent region connections (including diagonals) having been identified, we can begin moving our actors north to the battle during our fuzzy pathing phase.

4.5 Fuzzy Pathing Phase

We have an actor, a starting point, and a destination point, and now it's time to path! First, we need to determine the path region of our actor and that of the destination. From there, we hand this data over to a high-level (or fuzzy) pathfinding engine that computes a list of connected beacon points indicating the regions an actor must traverse to get to his destination. Luckily, this is not difficult if you're familiar with A*.

Fuzzy pathfinding engines work just like low-level pathfinding engines. In the case of A*, the algorithm finds the lowest cost route from a starting point to an ending point by examining adjacent regions, computing the costs for traversal, and flooding outward until a path is found. There are a few important differences, but overall, much of the A* engine design can use the same storage mechanisms [1], design [2], and optimizations [4] used in low-level A* engines. The big difference is that a path region's "neighbors" are not at the guaranteed left, right, up, down, and diagonal positions of a path region. Instead, they are all of a path region's adjacent neighbors. Think of regions as being nodes in a graph since a given region could contain ten regions on its right, and only one on its left.

Listing 4.3 demonstrates an A* engine update, and we see the adjacency iteration loop used in the A* machine [1]. We get the path region from the current A* node,

iterate over its adjacent regions, and if `RegionIsOpen()` method returns true, then we know our actor can legally move from the current region into the adjacent region. If the move is indeed legal, then the `CheckNeighbor()` method computes the costs of moving from one region to the other and places that node on the appropriate A* list.

Listing 4.3: Primary adjacency update loop for fuzzy pathing A* engine.

```
// Given our current node, get the path region from our A* node
theCurRegion = mCurrentAStarNode->GetPathRegion();

// Shown as foreach, insert your favorite loop iteration technique
foreach (PathRegion *theAdjacent, theCurRegion->GetAdjacents())
{
    // make sure this is not the same parent node.
    if (RegionIsOpen(theAdjacent, theCurRegion))
        CheckNeighbor(mCurrentAStarNode, theAdjacent);
}
```

Paths generated by A* are controlled primarily from two methods. First, the `RegionIsOpen()` method is used to determine if it's legal to traverse from one region to another. It handles not only the detection of path type validity, but checks diagonal movement as well. Lastly, the `GetRegionCost()` method indicates the cost of traversing from one region to another. This cost method has an enormous impact on path aesthetics and performance.

RegionIsOpen

Determining if an actor can move in a region goes back to our original design of path types. If a region is a forest, only our man-at-arms can move within it. If the region is an enemy wall, only our siege ram is valid. And as for our galley, it can only move in deep water.

The `RegionIsOpen()` method shown in Listing 4.4 not only needs to check if the new region is valid for the pathing actor, but whether, in the case of a diagonal move, the corners of the diagonal leap (Figure 4.3) are also valid.

Listing 4.4: This method determines whether an actor can move between regions.

```
bool AStarGraph::RegionIsOpen(PathRegion *inTo, PathRegion *inFrom)
{
    // If our actor can't walk in the new region, return false
    // this takes into account things such as...

    // The tile type is a gate, but it's locked. If that is the case,
    // it returns false.
    if (!mActor->CanMoveOnType(inToRegion->GetType()))
    {
        return false;
    }

    // Ask our current region to get the correct diagonals.
    // Note: there may be 2 or more juncture diagonal points
    // all with different regions at their diagonal edges.
    std::vector<std::pair<PathRegion *, PathRegion *> > diags;
    inTo->GetDiagonalBlockingRegions(inFrom->GetID(), diags);

    // If there are no blocks, it's passable.
    if (theBlocks.empty())
    {
        return true;
    }

    // See if this is a blockage.
    for (int i = (int) theBlocks.size() - 1; i >= 0; i--)
```

```
{
    // Can our actor walk on BOTH regions that are at the edges?

    // In other words, can it hop the diagonal legally by stepping
    // into the other regions momentarily?
    if (!(mActor->CanMoveOnType(diags[i].first->GetType()) &&
        mActor->CanMoveOnType(diags[i].second->GetType()))
    {
        return false;
    }
}

return true;
}
```

GetRegionCost

If an actor can move from one region to another, A* needs us to determine the traversal cost for the move. We're using oddly shaped regions, which could be a problem except that we normalized the sizes by using a world grid. For a basic cost, we use the world's grid. Notice that in Figure 4.3, we have grid sections with many regions within them. If we consider each black-bordered box has an (x, y) grid position, we'll use that to get the relative position for pathfinding. Listing 4.5 demonstrates how we compute the cost of traveling from a region to an adjacent region. Often, there are specific costs such as making it expensive (or less desirable) for an actor to move from land into a river.

Listing 4.5: Cost method for our A* fuzzy pathing engine.

```
unsigned long AStarGraph::GetRegionCost(PathRegion *inTo,
                                       PathRegion *inFrom)
{
    unsigned long theBasicCost = 0;

    // Special case for walls.
    if (inTo->GetType() == PathTileType::Wall)
        theBasicCost = 100;

    // If it's the same parent grid, we'll make the cost really low.
    if (inTo->GetGridPosition() == inFrom->GetGridPosition())
        return theBasicCost + 1;

    // Non-diagonal movement costs less than diagonal.
    if (inTo->GetGridPosition().
        IsDiagonalFrom(inFrom->GetGridPosition()))
    {
        return theBasicCost + 10;
    }

    // Diagonal movement costs more.
    return theBasicCost + 14;
}
```

Once the high-level pathfinder is done, we have a list of path regions. We should iterate through each one, extract the beacon points, and prepare our actor for low-level pathfinding.

High-level paths computed, our reinforcements are ready to move! Of course, if we followed the path points exactly as seen in Figure 4.4, our paths may not be as perfect

incrementally path along the way. When an actor approaches a beacon point, the actor paths to the next beacon point before arriving at the actual beacon point. If we displayed lines showing an actor's path, we would frequently see him cutting off the end of each beacon path while he moved and pathing ahead to the next beacon. That distance is something that a programmer must experiment with, as it affects both the time at which a low-level path calculation occurs (performance) and the overall aesthetics of a path. To give an example threshold for a grid resolution of 10×10 , we must meet one of two criteria to path to the next beacon: an actor must either enter a path region where his current beacon point exists, or he must be within 12 tiles of that beacon point. If one of those conditions is true, then we path the actor to the next beacon using our detailed path engine.

What we end up with is a nicely rounded path. Actors won't be walking to the middle of regions, only to turn and head in an unnatural angle to the next beacon. Simply put, we get a natural look, as shown in Figure 4.5. To the player, there is no evidence of a high-level grid-based path.

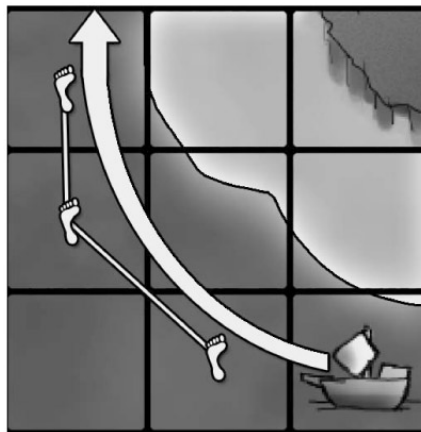


Figure 4.5: Beacon points serve as a guide to the detailed pathing engine. Note how the detailed path (round path with an arrowhead) does not go all the way to the beacon points.

4.7 Why Go Through All This Trouble?

One of the big advantages of our high-level path is in the constraints we can apply in low-level pathfinding. The A* algorithm is well known to suffer horribly from flooding problems. It often searches in the wrong direction and into concave spaces, and it ultimately checks many tiles that are unnecessary. Using our high-level path information, we can constrain our low-level pathfinder to only consider tiles inside our beacon-point (or their adjacent) regions. This keeps the low-level pathfinder from flooding outwards and backwards into areas that we know, at a high level, it doesn't need to check. This saves a huge percentage of A* loop iterations over the lifetime of the path.

The key to this optimization is its one-two punch of first using a high-level, fuzzy path to determine the correct, but ugly, route to an actor's destination. This first punch delivers incredible performance, something crucial for today's path-finding. Follow this up with our second punch, a series of low-level, detailed paths between beacon points. Suddenly, our ugly beacon-point paths look beautiful and intelligent. Separately, these path techniques have major strengths and weaknesses; however combined, strengths cancel each others weaknesses, and our pathfinding becomes unstoppable.

Acknowledgements

Special thanks to Rick Bushie for drawing our actors.

References

[1] Daniel F. Higgins. "Generic A* Pathfinding". *AI Game Programming Wisdom*, Charles River Media, 2002.

[2] Daniel F. Higgins. "Pathfinding Design Architecture". *AI Game Programming Wisdom*, Charles River Media, 2002.

[3] Daniel F. Higgins. "Terrain Analysis in an RTS—The Hidden Giant". *Game Programming Gems 3*, Charles River Media, 2002.

[4] Daniel F. Higgins. "How to Achieve Lightning Fast A*". *AI Game Programming Wisdom*, Charles River Media, 2002.

5

Environment Sound Culling

Simon Franco

The Creative Assembly

Overview

Each generation of game hardware brings with it new and exciting challenges for developers to tackle. One constant challenge with each hardware iteration is how to process data efficiently in real time. This must be done optimally to maximize hardware performance and deliver competitive results.

A number of techniques have been developed to efficiently handle spatial data for various systems. These include techniques such as using space partitioning structures to cull geometry that is not visible to the camera and using different AI complexity levels to reduce the processing time spent on distant or hidden characters.

A problem that receives less attention, however, is that of determining how to select and process real-time audio within a scene. There may be hundreds or thousands of permanent environmental sound sources in a scene in addition to the many transient sound sources that occur during gameplay, but only a small subset of them can actually be playing at any one time for performance reasons. This gem discusses a technique for efficiently reducing the complete set of sounds to the active set that is audible to the player.

5.1 The Problem

As our game environments increase in graphical detail, so too must our levels of audio detail to match the graphical representation of the game world. Numerous sounds are commonly layered to construct a game's environmental ambiance. This supersedes earlier techniques, which would have played a single stereo file to achieve the same goal. The advantage to playing multiple sounds, rather than a single audio clip, is that the sound closely matches the player's surroundings. For example, if the player was inside an old haunted house, there could be a grandfather clock, a television showing static, and wind howling through open windows. As the player moves through the house, the ambiance changes as the player goes from room to room. The player could change the ambiance by closing the windows or smashing the grandfather clock.

Environmental sounds to be played are selected from those near the listener. The listener represents a position and orientation in the game world from which the player is listening. Usually, this is either attached to the camera rendering the player's view of the world, or it uses some variation of the player's position and the camera's orientation.

The problem we must address is how to handle the multitude of sounds positioned within the game world. These sounds range from statically positioned continuous sounds such as fires and rivers, to more complex and dynamic audio events such as a crowd cheering on a fight, or a character interacting with a piece of animated geometry such as a lever.

All of these cases require that a sound emitter is placed within the world, either manually by a designer or as part of an automated process. Sound emitters are used to control how and when the sound is triggered, and most have a position from which the sound is triggered.

A sound emitter also contains a pointer to a sound event. Sound events are objects

containing information about how to play a sound, such as which wave file to play, volume, audible distance, pitch settings, and a priority. Priority checks are used as a method to select which audio channels are made available to the newly-requested sound event when there are no free channels available. If there are no channels available with a lower priority, then the requested sound event is not played.

While the game is running, each active sound emitter checks whether the listener is within audible distance (see Figure 5.1). If this test succeeds, and there are no free sound channels available, then a priority check against all currently playing sounds is performed. If both tests are passed, then the emitter can finally start playing its sound. The number of tests being performed each frame has increased as the number of sound emitters has risen in a typical game. Therefore, we need a fast method for rapidly rejecting large numbers of unsuitable sound emitters.

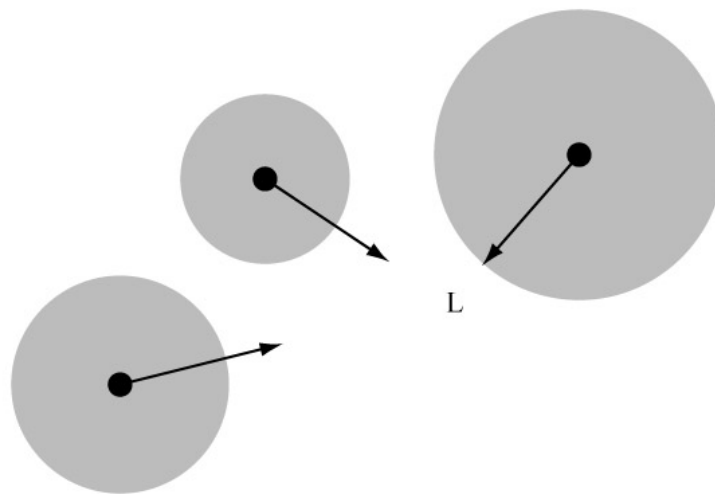


Figure 5.1: All sound emitters testing against the listener.

5.2 A Sound Culling Solution

We need to find a way to efficiently cull sound emitters that are not within an audible distance of the listener. We also need to cull any remaining sound emitters whose priority is too low to consider playing. The solution presented here involves the construction of a sound grid as a means of rapidly culling large numbers of sound emitters positioned within the game world. The sound grid is a two-dimensional grid parallel to the x - y plane that encompasses the entire game world. The grid is made up of equal-sized cells, and each cell contains an array of sound emitter lists. Each list within the array represents a different priority value, starting with the highest priority taking index 0 in the array. All sound emitters within a list have the same priority, matching that represented by the array's index. The sound emitters stored in these lists are those that are within audible range of that grid cell.

Since each grid cell contains lists of audible sound emitters, we only need to determine which cell contains the listener in order to know which sound emitters should be playing. This avoids having to perform complex searches for suitable sound emitters in real time.

We use a fixed cell size, rather than dividing up the space unequally, due to the nature of sound emitters. Sound emitters may be in physical proximity to each other, but have wildly differing audible distances. These different audible distances would cause any form of grouping to be potentially less efficient and result in more processing being used to determine which sound emitters are audible. Having a fixed cell size also allows for optimizations when determining which grid cell is occupied by the listener. The size of a grid cell can vary depending on your application, but too small of a size can lead to performance problems.

We take advantage of being able to group the sound emitters into priority lists, as

most applications will only use a limited priority range, typically between 5 and 10 different priority levels for environmental sounds.

Using a culling system such as a sound grid allows the audio system to rapidly cull thousands of potential sound emitting objects very quickly by only storing what can be heard within a given area of the world. By storing the sound emitters in matching priority lists, we can start by processing the highest priority list and quickly bail out of our sound emitter processing if we have run out of free sound channels and have reached a priority level that is too low.

Listing 5.1 shows an example sound grid cell along with an example sound emitter and linking class used to bind them. We use instances of the `SoundEmitterLink` class within the `SoundEmitter` class to form the linked list connecting up sound emitters of matching priority within a cell. The `SoundEmitter` class contains the `m_cells_touched_array` member, which is an array of `SoundEmitterLink` objects. The array's size is set on constructing the sound emitter object, and should be the maximum number of cells that could be touched by that `SoundEmitter`.

Listing 5.1: This pseudocode shows an example sound grid cell and sound emitter.

```
struct Cell
{
    SoundEmitterLink    *m_emitter_list[MAX_PRIORITY_LEVELS];
};

struct SoundEmitterLink
{
    Cell                *m_cell;
    SoundEmitter        *m_parent;
    SoundEmitterLink    *m_prev;
```

```
    SoundEmitterLink    *m_next;
};

struct SoundEmitter
{
    Vector              m_pos;
    SoundEvent          m_sound_event;
    SoundHandle         *m_sound_handle;
    SoundEmitterLink    *m_cells_touched_array;
    int                 m_num_cells_touched;
    bool                m_active;
};
```

5.3 Constructing the Sound Grid

The sound grid is constructed using data from both static and dynamically moving game objects. A game object is an object created by the game that has information about the sound event it wants to play and where the sound should be positioned.

The sound grid is first constructed using the static game objects. We process each static game object present within the game world only once when the game's level is loading. We use the audible distance and position of each game object to determine which grid cells its sound emitter touches (see Figure 5.2). We construct a single sound emitter for that game object and setup its `m_cells_touched_array` member for the number of cells within audible distance. For each of those grid cells within audible range, we use a free element in the `m_cells_touched_array` member to form a link between that cell's appropriate priority list and the newly constructed emitter. We return the pointer for the newly-constructed sound emitter object back to the game object to optionally store in case it needs to later make modifications to the emitter's state.

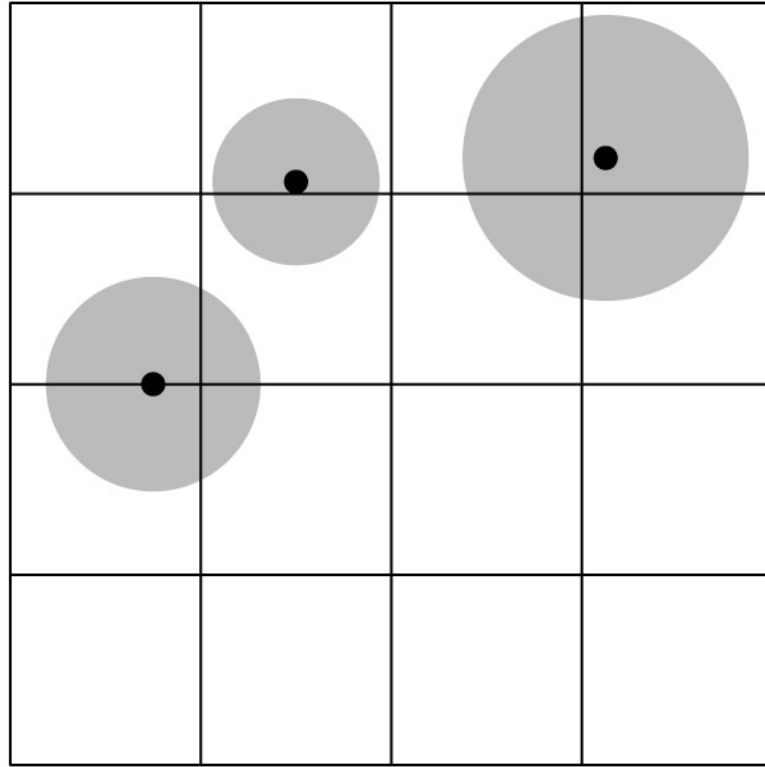


Figure 5.2: Sounds emitters using their audible distance to determine which grid cells they touch.

Some sound emitters do not occupy a single fixed position in the game world. For example, you may have a spline representing a river that is to have a sound emitter placed at the position nearest to the listener on the spline. Another example may be that the game world contains a forested region inside of which you wish to play a bird chirping sound at a randomly determined position. For these cases, we construct the sound emitter as with static sounds, but do not add it to any grid cells. We treat these as dynamic sound emitters, which will be changing which grid cells they belong to as the game progresses.

5.4 Processing the Sound Grid

Once per frame, the listener's position is converted from a world-space position to the particular cell covering that space within the sound grid. This is so we can retrieve from that cell the lists of audible sound emitters, which we'll need to try playing. The sound grid retains a copy of the sound emitters selected from the cell visited on the previous frame. This list is referred to as the active list, as it contains the list of all sound emitters that should be actively playing. The active list has a fixed maximum size, matching the maximum number of sounds that can be played at any one time.

Listing 5.2 shows the process for building up the active list. The first processing phase is to mark all sound emitters in the active list for removal. Each element in the cell's emitter list, up to the maximum number of playable sounds, is checked against the active list of emitters. If an emitter in the cell's list points to the same emitter in the active list, then that emitter's removal flag is cleared. We only test up to this number of elements as additional elements could not be played.

Listing 5.2: This pseudocode shows how we build the active list and stop playing invalid sounds.

```
void SoundGrid::buildActiveList()
{
    // phase 1.
    set_all_active_list_emitters_for_removal()

    for (priority = 0; priority < MAX_PRIORITY_LEVELS; ++priority)
    {
        for each emitter in the cell.m_emitter_list[priority]
        {
            for each active_emitter in the active_emitter_list
            {
```

```
        if (emitter == active_emitter)
        {
            emitter.set_unremoved()
            break
        }
        else if (emitter.m_priority > active_emitter.m_priority)
        {
            // Emitter can't be on the active emitter list as
            // we've gone past its priority.
            break
        }
    }
}

// phase 2.
for each active_emitter in the active_emitter_list
{
    if (emitter.is_removed()) emitter.stop_any_playing_sounds()
}

// Finally copy the cell's emitter list to the active emitter list.
// Start with the highest priority list and progress to the lowest
// priority list until we run out of sound emitters or fill up
// active_emitter_list
copy_list_to_active_list(active_emitter_list, cell_emitter_list)
}
```

The second phase is to then run through the active list and stop playing any emitters that are still marked for removal.

Now, the only sounds currently being played are those that are in the cell's emitter

list and that were in the active playing list. We finally copy the cell's emitter list into the active list. Again here we only copy up to the maximum number of playable sounds to avoid redundant data.

Once per frame, we process the active list of emitters to see if any of them are either in an "on" state but not playing, or are in an "off" state but are playing. This is shown in Listing 5.3. In Figure 5.3, we show how only one sound emitter is valid when using a sound grid.

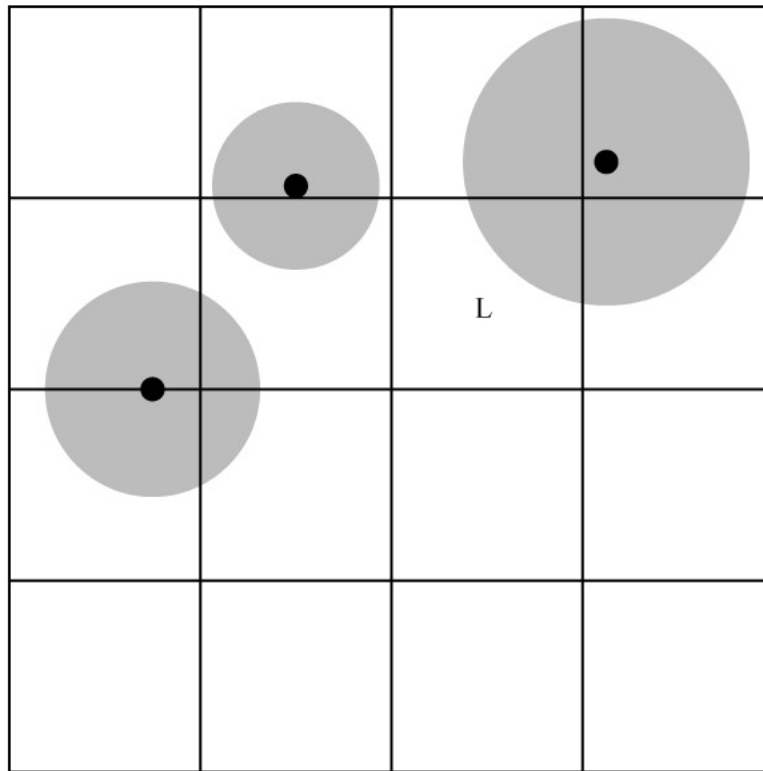


Figure 5.3: The cell occupied by the listener is only within the radius of one sound emitter.

Listing 5.3: This pseudocode demonstrates how the active emitter list is processed.

```
void SoundGrid::updateEmitters()
{
    for each active_emitter in the active_emitter_list
    {
        if (emitter.is_on())
        {
            if (emitter.not_playing())
            {
                if (engine.sound.get_num_channels_free() > 0)
                {
                    emitter.play_sound()
                }
                else if (emitter.m_priority >
                    engine.sound.lowest_priority())
                {
                    emitter.play_sound()
                }
            }
            else
            {
                emitter.update_sound()
            }
        }
        else if (emitter.is_playing())
        {
            emitter.stop_sound()
        }
    }
}
```

More on Sound Emitters

As mentioned earlier, a pointer to the newly-constructed sound emitter object is returned once a game object has had its sound emitter constructed. The purpose of this is to allow the game to play or stop a sound emitter by changing its "on" flag. This flag is a required part of a sound emitter, as not all game objects require that their sound emitter play continuously. Examples of this are when the player has set fire to an object or the game object is coordinating the sound being played during particular frames of an animated piece of geometry.

How to Handle Dynamic Sounds

Some game objects need to perform an update once per frame on the position of their sound emitter. For example, you may have a sound that moves along a predefined path and need to update where it is on that path each frame. To achieve this, we take advantage of the sound grid's structure to allow for dynamic sounds. During each frame, a game object such as a car can move its sound emitter. To do this, it must first have the sound emitter remove itself from all the grid cells it was previously touching. The sound emitter is then moved to its new location in the world and re-added to the sound grid. This must happen before the sound grid is processed and the active emitter list is built. Listing 5.4 shows an example algorithm for inserting the sound emitter's link to the head of a cell's emitter list, and Listing 5.5 shows the matching removal algorithm.

Listing 5.4: Sample insertion routine for dynamic sounds.

```
/*
 * This inserts the SoundEmitter to the head of the corresponding
 * priority emitter list for this cell.
 */
void SoundEmitter::add_emitter_to_cell(cell)
{
```

```
int index = m_num_cells_touched
int priority_level = m_sound_event.m_priority

++m_num_cells_touched

m_cells_touched[ index ].m_cell = cell
m_cells_touched[ index ].m_prev = null

// Set the next element to what the first element (if any)
// was pointed to in the cell's list.

m_cells_touched[index].m_next = cell.m_emitter_list[priority_level]

// Was there something at the head of the emitter list? If so,
// have its prev link point to this entry.

if (cell.m_emitter_list[priority_level])
{
    cell.m_emitter_list[priority_level].m_prev =
        m_cells_touched[index]
}

// Finally set the head of the cell's emitter list
// to be this sound emitter link.
cell.m_emitter_list[priority_level] = m_cells_touched[index]
}
```

Listing 5.5: Sample removal routine for dynamic sounds.

```
// This removes the SoundEmitter from all cells its touched.

void SoundEmitter::remove_emitter_from_cells()
```

```
{
    int priority_level = m_sound_event.m_priority
    for (index = 0; index < m_num_cells_touched; index++)
    {
        if (m_cells_touched[index].m_next)
        {
            m_cells_touched[index].m_next->m_prev =
                m_cells_touched[index].m_prev
        }
        if (m_cells_touched[index].m_prev)
        {
            m_cells_touched[index].m_prev->m_next =
                m_cells_touched[index].m_next
        }

        // Check if this was the head node of the linked list. If so
        // change the head node to point to the next node (if any).

        cell = m_cells_touched[index].m_cell
        if (cell.m_emitter_list[priority_level] =
            m_cells_touched[index])
        {
            cell.m_emitter_list[priority_level] =
                cell.m_emitter_list[priority_level].m_next
        }

        // Clean this SoundEmitterLink up.
        emitter.m_cells_touched[index].m_cell = null
        emitter.m_cells_touched[index].m_prev = null
        emitter.m_cells_touched[index].m_next = null
    }
    emitter.m_num_cells_touched = 0
}
```

5.5 Supporting Multiple Listeners

Some games need to support more than one player using the same television or monitor. This requires that the display is split in some fashion to show both players' view of the world. As well as having both players' view being processed by the same game console, we must also divide up the sound channels to represent what each player is hearing. The sound grid can be modified to support this with a few changes to the way the active list is built:

- We first construct an active list for each listener present in the game.
- Once this has been done for all listeners, we copy the highest priority sounds from each list into the master active list.
- We then make sure any sounds that were in any of the listeners' active lists and didn't make it into the master active list are not playing.

5.6 Extensions

The sound grid is one possible solution to culling sounds occupying known world positions. While the implementation discussed in this gem only constructs a two-dimensional grid, this technique should work without change for most types of game worlds. If your game contains a high number of vertical sounds occupying a nearby space in x and y , then the process can be extended to allow for culling sound emitters in the z dimension. Either you may wish to have multiple sound grids, with each one at a different z height, or expand into a 3D sound grid.

Additional future work can be performed on the sound grid, such as embedding additional information about the grid cell or sound emitters, such as which reverb effects to apply to the emitters, or whether we need to apply a filter (due to an

obstruction between the sound emitter and listener). Dynamically generated sounds, such as gunfire, may also be able to use information contained in the sound grid to calculate any filtering that needs to be applied, avoiding the need to perform expensive tests in real time. Also, additional optimizations could be made to the sound emitter linked lists. We could, for example, add a head and tail node to each linked list and thus remove the conditional tests surrounding the insertion and removal of a sound emitter to a grid cell.

6

A GUI Framework and Presentation Layer

Adrian Hirst

Weaseltron Entertainment Limited

Overview

Graphical user interface (GUI) design is an often overlooked and under-resourced part of game development, yet it is responsible for the look and feel of a game to the user as well as its all important first impressions. The user interface needs to be quickly and dramatically adapted continuously throughout a product's development cycle in order to reflect modifications to virtually any other part of the game.

Relatively little literature exists for GUI presentation code, and finding samples of rigorously tested source code is difficult. This gem provides a brief introduction to GUI systems and documents a proven, current, flexible, and working system that can provide the first step for in-game and tools-based systems. Drop-in source code is provided and should prove instantly useful for everyone from the student to the seasoned professional.

6.1 GUI Systems

Like many in-game systems, the difficulty of arriving at the finished product is in

being flexible enough to respond to constant changes. The iterative nature of game development means that a game is constantly changing. For every change to scoring systems, game mode, or indeed any gameplay element, a subsequent change is most often required to the presentation layer, ensuring that the user is still clearly presented the relevant and required information they need to play the game. Iterative game design impacts many aspects of a game throughout its development, all of which needs to be fed back to the user, a task that falls to the presentation layer.

A common and sensible approach is built around creating a set of solid, stable controls, or components that can be used and combined over and over in various ways. When requests come in to the presentation team for a new screen, or there's a new gameplay mechanic that requires a screen to be rewritten, a familiar set of text boxes, buttons, menus etc. can be dropped into place.

A GUI system should include an editor for artists to create and position components, textures, and text in the best way. Too often, and particularly on smaller projects, rather than creating an editor for this task, the job of positioning these screen elements falls to a programmer, who has to type the coordinates by hand into a text file. This is unacceptable for anything but the smallest of games and interfaces.

Localization issues always create their own problems in user interface design. Translation of games to multiple languages inevitably leads to text strings of varying length, often occupying more screen space than initially allocated. Even seemingly innocent changes in the wording of key phrases, or even simple modifications in capitalization or grammar, can cause text strings to overlap the area of screen designated. The new word is likely to be longer in another language—the German language generally being the most verbose. An essential feature of using an editor to design presentation screens is being able to preview static text in all languages to check for such issues.

Console platform holders typically have their own set of technical requirements that must be fulfilled before the game is accepted. Without going into too much detail, these range from new screens, game modes, network requirements, controller configurations, and menu options to screen resolutions, drawable areas of the screen, and issues of text legibility.

Existing Solutions

Several middleware solutions exist for creating front ends that provide full feature sets, including WYSIWYG editors, runtime components, custom animation, scripting, and Adobe Flash support. These prove compelling where the budget allows. Such tools provide support that empowers artists and designers to create the best looking and most advanced GUI systems with as little programmer involvement as possible.

Support for these features comes at a cost, though, and some implementations can add significant memory and CPU overhead, with some Adobe Flash implementations in particular being typically resource-heavy. Smaller-scale products, however, often do not require such feature-rich implementations and can trade features for performance and flexibility.

6.2 Design Patterns: Model View Controller (MVC)

A common problem, particularly evident with GUI systems, is that over time, last minute hacks and "temporary" bug fixes lead to strongly coupled code, leaving changes to one area of the system to cause unintended effects elsewhere and also making it difficult to refactor. The model-view-controller (MVC) design pattern aims to ensure only a loose coupling of elements by separating the framework into three distinct constituent parts to be maintained independently:

- The *model* refers to the actual data that we represent on the screen. For example, this could be the game time or the number of lives remaining.
- The *view* concerns itself only with the visual representation and rendering of that object. In our previous example, perhaps we might display an analogue clock to represent the time remaining or an icon for every life remaining.
- The *controller* refers to how the object interacts with the game, user input, and general state, system, or game logic.

MVC has gained popularity, now being a major contributing concept in many of the larger GUI and system frameworks, from Cocoa to Qt, MFC, and the current Windows Presentation Foundation.

Taking our previous timer example, the *controller* would refer to our game update loop; being responsible for determining the current time and passing a `TimerUpdate` message to the timer *model* object, which in turn updates its internal representation. The *view* will typically have its own internal description, but will update that based on information inside the *model* itself.

Implementing MVC does require a change in thinking, as the three constituent parts are unaware of the others' inner workings and unable to alter each other's data so communication relies on sending messages. This could just take the form of calling a function on another object. In larger systems, though, it is beneficial to ensure a greater distinction between components by using event queues and traditional message passing systems.

Depending on the technique used, this message passing can add a substantial amount of overhead to the system, reducing its flexibility. Choosing the correct method by which these messages are passed is crucial, and worthy of its own Gems topic.

6.3 A GUI Design

The goal of this gem is to present a simple and extensible drop-in GUI module whose target audience ranges from the student or smaller indie studio to the experienced professional wanting to quickly add some user interface interaction to their game without devoting too much of their time to reinventing the wheel. Therefore, the system must exhibit the following attributes:

- **Modular.** As a drop-in replacement, it must have the minimum possible number of external API dependencies.
- **Lightweight and flexible.** It should be adaptable to as many platforms as possible, have bindings to specific areas such as various input types, and rendering or audio code should be kept as minimal as possible. This guarantees the highest level of portability.
- **Both programmer and artist-driven.** For many smaller products, it is often feasible (if not fun) to hand-code our own front end design. While this exposes us to the dangers of localization, minor font changes, and rewordings, it can sometimes be the quickest way of creating a usable GUI. This system allows programmatically created GUI screens where necessary, and it leaves scope for data-driven designs from an external tool at a later date.
- **Extensible.** The code provided should give the user a solid base from which further, more advanced controls may be produced with the least amount of effort.
- **Object oriented.** Object-oriented programming tends to lend itself particularly well to GUI objects with inheritable behaviors and data sets.
- **Localization-ready.** All text rendering must be able to handle international character sets.
- **"Boilerplate code"-minimizing.** It should encapsulate common programming tasks so the user can concentrate on adding content rather than repetitive functionality.

A Little Code ^[1]

First, let's look at the basic `GuiComponent` class shown in Listing 6.1. This is the base class from which all other GUI components are derived, and it contains very little specific information or functionality other than it is an object, it has a visual representation that exists somewhere in a hierarchy, and it has a little information about whether it is active and whether it is visible.

Listing 6.1: Our `GuiComponent` base class.

```
class GuiComponent
{
public:

    GuiComponent();
    virtual ~GuiComponent();

    virtual TypeId GetTypeId() const = 0;
    static TypeId GetStaticTypeId();

    uint32_t GetId() const;

    void SetupGuiComponent(const String& name, bool active = true);
    void SetVisual(Visual *visual, bool makeVisible = true);
    void SetPosition(const Vec3& pos);

    virtual void Update(Time& timeDelta) = 0;
    virtual void Render(Renderer *renderer);

private:

    static TypeId s_TypeId;
```

```
String      m_Name;  
uint32_t    m_Id;  
Frame       m_Frame;  
Visual      *m_Visual;  
  
// Is the GuiComponent taking input and being updated etc.?  
bool        m_Active;  
  
// Default to invisible false until we have a valid GFX::Visual.  
bool        m_Visible;  
};
```

The `GuiComponent` class contains two simple boolean variables that determine whether the object is considered active and whether it is visible. The distinction here is that an object could be visible on screen, but not active in that it does not respond to input or update itself. Likewise, an object could be actively being processed, but not visible on-screen.

A string is stored to keep a human-readable name for the object. This is largely used for debugging, but also used to generate the CRC of this name, which is stored in the `m_Id` variable for fast access and comparison/finding. This can also be used by tool editors and loaders for referencing objects.

The `GuiComponent` also contains a `Frame` node that specifies a local translation to apply to this object, as well as its position inside a hierarchy. This allows us to attach together `GuiComponents` into groups which can then be moved together as a whole.

The model contains a single pointer to a `Visual` object containing rendering information. I consider this to be a loose abstraction of the model and view of the object, but it serves our design goal of being quick and flexible well, meaning that it abstracts much of the rendering code away while still allowing us to have an entry point for

patching any rendering information, such as state or material changes, where necessary, for those last-minute fixes. For medium-sized or larger systems, the coupling between the model and view side of the object would need to be looser. The `Render()` function can simply make the following call:

```
renderer->Render(m_Visual);
```

This can be overridden via virtual function in derived classes where required.

This `GuiComponent` class allows us to extend trivially to create a basic `GuiSprite` class, as shown in Listing 6.2. This simplest of examples just adds an instance of a `VisualSprite` class and calls the renderer to draw it. Here, the `VisualSprite` itself contains most of the information required to render the sprite, which is set up inside the `GuiSprite::SetupGuiSprite()` function. Calling the `GuiComponent::SetVisual(&m_Sprite)` function inside the `SetupGuiSprite()` function ensures that the base class render function performs all of the rendering required for objects of this type.

Listing 6.2: A sprite class.

```
class GuiSprite : public GuiComponent
{
public:

    void SetupGuiSprite(const Vec3& pos, Texture *texture,
        Color& sprite, float width = 0.0F, float height = 0.0F);

    virtual void Update(Time& delta);

private:
    VisualSprite    m_Sprite;
};
```

Likewise, a `GuiTextItem` class can be derived that stores a pointer to text and rendering parameters, as shown in Listing 6.3. In the full code sample on the accompanying CD, a `BMFont` class stores glyph-based information and texture data loaded from files exported from the AngelCode `BMFont` library [2].

Listing 6.3: The `GuiTextItem` class.

```
class GuiTextItem : public GuiComponent
{
public:

    void SetupGuiTextItem(wchar_t *text, FONT::BMFont *font);

    float GetTextWidth() const;
    float GetTextHeight() const;

    void SetText(wchar_t *text);
    void SetAlignment(GuiTextHAlign hAlign, GuiTextVAlign vAlign);

    void SetColor(const Color& color);
    virtual void Update(Time& delta);

private:

    float          m_xScale;
    float          m_yScale;

    VisualText     m_VisualText;
};
```

Note that the `GuiTextItem` class only accepts wide character strings to display text. These multi-byte characters enable us to display characters from multiple languages,

though various platforms may endian-swap the order of the bytes. Two functions get the calculated width and height of the text based on the text and the font. This becomes useful when we want to allow a piece of text to be selected with a pointing device.

Extending our System

Unfortunately, GUI systems are required to perform more than simply displaying sprites and text, so let's look at adding another important class, `GuiSelectable`. Shown in Listing 6.4, the `GuiSelectable` class is a parent class to menu items, clickable icons, selectable text, and anything else that might be highlighted and/or selected. The purpose of this class is to provide an interface that responds to two major activities—being highlighted (for example, when a user points their mouse, stylus, or other pointing device over the component) and being selected (for example, when the pointing device is clicked on this item or the "select" button is pressed). The device type being used is irrelevant, as highlighting and selecting are actions common to on-screen navigation, whether by keyboard, joystick, stylus/touch-screen controllers, remote pointing devices, etc.

Listing 6.4: A base class for selectable components.

```
class GuiSelectable : public GuiComponent
{
public:

    bool IsSelectable() const;
    virtual void SetSelectable(bool selectable);

    bool IsHighlightable() const;
    virtual void SetHighlightable(bool highlightable);

    GuiHotSpot *GetHotSpot();
```



```
void SetHotSpot(GuiHotSpot *hotSpot);

// Process what to do when the users focus is
// in the specified position.
virtual bool SetFocus(float x, float y);

// Test to see if the specified position is within the HotSpot.
virtual bool IsInHotSpot(float x, float y);

// Virtual function called when the item is selected and returns
// whether it is possible to select this item.
virtual bool OnSelect();

// The GuiSelectable is highlighted
// (mouseover, menu item selected).
virtual bool OnHighlight(bool highlighted);

virtual void Update(Time& delta);
private:

    GuiHotSpot      *m_HotSpot;

    bool            m_Highlightable;
    bool            m_Selectable;
};
```

A `GuiSelectable` object contains a reference to a `GuiHotSpot` item, which holds information about the area of the screen that, when clicked on, makes the `GuiSelectable` object active. Typically, this is a rectangular region roughly the same size as the icon or text of the item itself. Derived classes can describe their own circular or polygonal areas as long as the `IsInside()` virtual function is overridden. `GuiHotSpot` objects retain a pointer to a `Frame` object so that the object tracks with the

`GuiComponent` object itself.

For pointer-based devices, determining the highlighted status of a `GuiSelectable` object is achieved by passing the current on-screen cursor coordinates to the `SetFocus()` function. If, after querying the `m_HotSpot` variable, it's determined that the highlighted state should change, then the virtual `OnHighlighted()` function is called with the new desired value. Again, we can override this virtual function in derived classes to update its own state, and therefore, its visual representation.

For button-based controllers (primarily keyboards and joypads/sticks), the focus is likely to be more bespoke, perhaps being transferred by tabbing through the various highlightable components, or in the case of menus, using the directional pad on an input device. We revisit this topic later.

`GuiSelectable` objects can be told that they are not highlightable. While in this state, they are not able to become active like other `GuiComponent` objects unless explicitly made highlightable again. This is most likely to be used for components that are either just not available or are not available yet. For example, there may be a currently hidden game mode option.

Likewise, it may not be possible to actually select some items, even though they are highlightable. Perhaps, for example, moving our cursor over the currently unavailable last level on a level select screen highlights or animates that component, but the user is still not able to select it. Objects being selected have the `OnSelect()` virtual function called on them, which returns whether the selection succeeded. This is then handled by the current state.

But What About the Menus?

Okay, so we've got the ability to place objects around the screen wherever we like, but wouldn't it be nice to put a collection of these items into a menu? Enter the `GuiSelectableGroup` class, shown in Listing 6.5. A `GuiSelectableGroup` object can

be thought of as a menu, with a collection of pointers to `GuiSelectable` objects that make up its menu items. This allows us to move common code for highlighting and selecting groups of objects into a single interface. At its simplest, the `GuiSelectableGroup` class allows us to highlight objects by array index, by `GuiComponent` name, or by ID. It also allows us to simply highlight the next or previous item in the current group, skipping unhighlightable items and wrapping around the list, if required.

Listing 6.5: A `GuiSelectableGroup` handles the selection of many `GuiSelectable` objects.

```
class GuiSelectableGroup : public GuiSelectable
{
public:

    // Add item and set the attached item's m_Frame's parent to us?
    virtual void AddItem(GuiSelectable *item, bool attachFrame);

    // Remove all items from list and detach any attached Frames.
    virtual void Clear();

    // Highlight the next/previous item skipping any unhighlightable
    // items and wrapping where necessary.
    virtual bool HighlightNext();
    virtual bool HighlightPrevious();

    // Highlight an item in the list by index, component name
    // or Id (name CRC).
    virtual bool HighlightIndex(int index);
    virtual bool HighlightItem(const String& name);
    virtual bool HighlightItem(uint32_t id);
```

```
// Find an item in the GuiSelectableGroup either by name
// or Id (name CRC)
virtual int FindItemIndex(uint32_t id);
virtual int FindItemIndex(const String& name);

virtual GuiSelectable *FindItem(uint32_t id);
virtual GuiSelectable *FindItem(const String& name);

// Get the currently highlighted object (if there is one).
GuiSelectable *GetHighlighted();

virtual bool SetFocus(float x, float y);

// Test to see if the specified position is within
// the GuiComponent's HotSpot.
virtual bool IsInHotSpot(float x, float y);

// Virtual function called when item is selected and activated.
virtual bool OnSelect();

// Virtual function called when the GuiSelectable is highlighted
// (mouseover, menu item selected).
virtual bool OnHighlight(bool highlighted);

// Update all the items in this GuiSelectableGroup.
virtual void Update(Time& delta);

private:

// A static-sized array of pointers to GuiSelectable items.
GuiSelectable *m_Selectables[GUI_SELECTABLEGROUP_MAXITEMS];
int m_NumSelectables;
```

```
int          m_Highlighted;  
bool         m_Wrapped;  
};
```

Adding an item to a `GuiSelectableGroup` object is done via the `AddItem()` function, which provides the option of automatically attaching the new item's `Frame` to this one. If used as a menu, this attachment means that all items in the menu can be moved together, perhaps animated onto or off of the screen just by animating the *x* component of the root node matrix. Note also, that after adding items to an empty list, if we want one highlighted, we still want to manually do that, or call `HighlightNext()` to highlight the first available. This is typical for a console or keyboard-based control system.

This class conforms to the *composite* design pattern, as the `GuiSelectableGroup` class itself derives from the `GuiSelectable` class. This straightforward, but powerful distinction means that it inherits the ability to be selected, to be highlighted, and importantly, to be included in a list inside another `GuiSelectableGroup` object. This is useful for a number of situations—for example, with an options screen where the main items run vertically, selected by up/down on the joypad, but where some items provide multiple options to choose from, such as selecting low, medium, or high for graphics detail, stereo or mono for sound, etc., which run horizontally.

In conforming to the composite design pattern, the `GuiSelectableGroup` class is responsible for managing calls through the `GuiSelectable` virtual functions too. `Update()` and `Render()` calls are passed through to all contained `GuiSelectable` objects, the `OnSelect()` function determines the currently highlighted item and (if present and active) calls the relevant `OnSelect()` function only on that item. Functions such as `OnHighlighted()` and `SetActive()` follow a similar pattern, though it is worth browsing the code to see the details of when objects become active or not.

An Example in Use

The code excerpt shown in Listing 6.6 uses a type `GuiSelectableText`. This is a `GuiSelectable` object containing just a `GuiTextItem` object. It shows how quickly the class structure can create an on-screen menu. A `RenderComponents()` function trivially reduces to just the following:

```
Menu.Render(renderer);
```

Listing 6.6: Setting up and updating of an example state.

```
GuiSelectableText MenuItems[MENUITEM_COUNT];
GuiSelectableGroup Menu;

void SetupComponents()
{
    Menu.SetPosition(Vec3(512.0F, 480.0F, 0.0F));

    for (int item = 0; item < E_MENUITEM_COUNT; ++item)
    {
        MenuItems[item].SetupGuiComponent(s_ComponentNames[item]);
        MenuItems[item].SetupGuiSelectableText(s_ComponentText[item],
            Resource_GetFont());
        MenuItems[item].SetPosition(menuItemPos[item]);

        Menu.AddItem(&MenuItems[item], true);
    }

    MenuItems[E_MENUITEM_UNLOCK_ME].SetHighlightable(false);

#ifdef CONTROLLED_BY_POINTER_ONLY
    Menu.HighlightNext();
#endif
}
```

```
}

// The update loop then reduces to this:
void UpdateState(Time& delta)
{
    Menu.Update(delta);

#ifdef CONTROLLED_BY_POINTER_ONLY

    if (m_Menu.SetFocus(pointerX, pointerY))
    { // If we're only using the mouse, and we're not over any
      // other item, unhighlight the currently selected one.
      m_Menu.HighlightIndex(GUI_SELECTABLEGROUP_UNHIGHLIGHTED);
    }

#else

    m_Menu.SetFocus(x, y);
    // Now loop through all the controllers checking their actions.
    for (all active controllers)
    {
        if (Controller.IsPressed(DPAD_UP))
            Menu.HighlightPrevious();
        else if (Controller.IsPressed(DPAD_DOWN))
            Menu.HighlightNext();

        if (Controller.IsPressed(SELECT) && Menu.OnSelect())
            ProcessSelect();
    }

#endif // CONTROLLED_BY_POINTER_ONLY
}
```

Adding virtual functions `SetupComponents()`, `UpdateComponents()`, `RenderComponents()`, and `UpdateInput()` to the base state system helps build a framework that can be extended and keeps a common interface throughout the GUI screens as well as separate the GUI components code from other state-based code. This also helps to distinguish the model, view, and controller portions of the code. This can be seen better in the source code on the accompanying CD, though a full explanation is outside the scope of this gem.

^[1]Note that the code in this gem is a shortened version of the full code available on the accompanying CD. Simple `Get()` or `Set()` and other ancillary functions have been removed for the sake of brevity.

6.4 And Finally

The objects presented in this gem forms just the beginning of a GUI system. Included on the accompanying CD is an example test application implementing these and other `GUIComponent` objects by demonstrating a variety of example states. Also included is a workable `StateMachine` class, a very basic OpenGL renderer built upon SDL, test font library, input library, basic MATHS components and various foundation, or utility classes and anything else needed to put together an implementation of a basic GUI framework.

To take this work a step further, an element of data-driven design needs implementing. In particular, we need an editor for artists to take control over the look and feel of the presentation.

An up-to-date version of the code is available on the Weaseltron website at <http://www.weaseltron.com/WeaselGui>. The code is distributed under the LGPL license, meaning it's free to use in commercial products.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] AngelCode BMFont. <http://www.angelcode.com/products/bmfont/>

7

World's Best Palettizer

Jason Hughes

Steel Penny Games, Inc.

7.1 Palettes? Whatever for?

Back in the old days of fixed-function graphics cards and limited VRAM, palettized textures were one of the earliest forms of compression, at the cost of color accuracy. However, as VRAM increased on video cards, and graphics chipsets began moving toward programmable shaders, palettized images gave way to S3TC/DXTC (and better) compression that generally give the same memory savings with better performance and color depth.

So, why use palettes? Perhaps you've heard of its highbrow cousin, vector quantization? It may seem like a crazy thing to need in this day and age, but quantization codebooks (palettes) are useful in many contexts, not just with images. Some examples: S3TC/DXTC is not a good choice for all images, particularly cartoonlike graphics with large regions of solid color divided by sharp edges. Finely detailed pixel art ends up as a blocky mess. Also, some GPUs support individually indexed vertex attributes such as normals, colors, and so on, but are limited to either 8-bit or 16-bit indices. This is effectively a palette of values. Similarly, with shaders on modern systems, textures could be used to replace certain vertex attributes to simplify

the number of vertex formats involved in setting up a flexible engine, and compression of those textures could be effective with a smaller 1D lookup table texture having a limited set of values. Normal maps might also be good candidates for palettizing, depending on the complexity of the surface. S3TC/DXTC block artifacts can show up in lighting on some surfaces whereas palettized versions would at worst show banding, but not blocking artifacts.

Finally, there are the occasional obscure but amazing palette tricks that old game dogs know. One such trick put palettized vertex colors to excellent use. The general idea was to vertex-light the world in N different phases of day and night, then compute a $3N$ -dimensional palette where each "color" in the source image was effectively a stack of N RGB triplets. Every vertex in the world was given one vertex color palette index that was the stacked color in the final palette. So, without changing any vertex data, simply by interpolating the colors inside the palette from one set of RGB to the next, vertices appear to go through a smooth change in time of day.

For reference, Figure 7.1 shows how the WBP compares to Photoshop, with error diffusion disabled so we can clearly see the quality of the color choices.

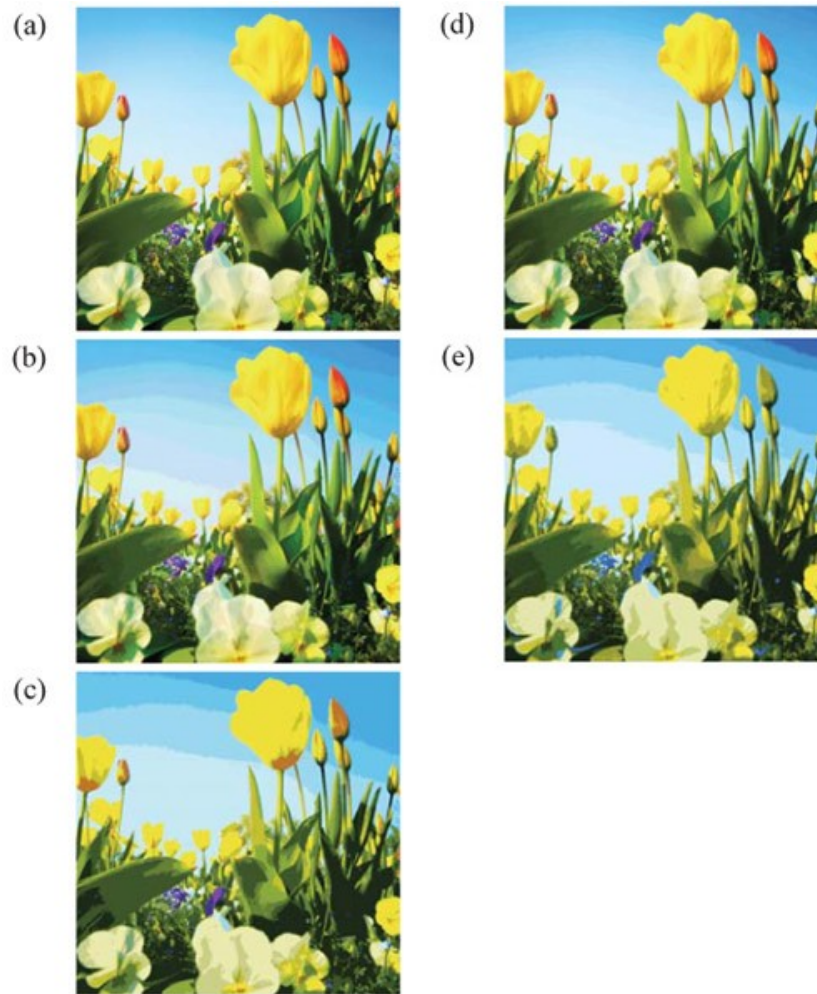


Figure 7.1: (See also Color Plates.) (a) Flowers source image using 100,162 colors. (b) Photoshop, 256 colors. (c) Photoshop, 16 colors. (d) WBP, 256 colors. (e) WBP, 16 colors. (© Dundanim/Dreamstime.com)

7.2 Understanding Quantization

For such a simple concept, *reduce the cardinality of a dataset from M samples to a fixed number N that minimizes error*, there are a surprising number of tough lessons to be learned when designing a quantizer. Some interrelated questions have to be answered that vary with the data and the situation to which they're applied:

- How do you compute a representative value for multiple samples?
- What is a good metric for measuring error between samples and their representative values?
- Are all aspects of the input data equivalent, or is there a need for weighting some channels or computing error differently for them?
- Is it better to start with M representative values and merge them?
- Is it better to start with one and subdivide until reaching N ?
- What about stochastic processes that gradually improve the fit until some measure is satisfied?

If you study the research on palettizing algorithms and vector quantization, there are many trade-offs that various approaches make, such as tools performance against runtime quality, code complexity against ease of implementation, perceptible error against numerical error, etc. What is important is that you understand that this is effectively a k -means clustering problem, which is NP-hard even for finding two palette entries for a data set, much less N [1]. So, for any algorithm to be practical, there will be assumptions made, and possibly inadequate results for specific inputs. It is therefore important to try not to solve too general a problem and only worry about large sources of error as long as the results are positive.

7.3 Hard-Earned Lessons

The best experiences (in programming) are those of failure. As Charles Kettering once said, "One fails forward toward success." Here are a few observations after designing several algorithms that did not produce satisfactory results. Others may have better luck or better ideas, but the following were important lessons I learned:

- Colors and other attributes can be unified nicely by converting them all to floating point numbers and passing them around as arrays or `std::vector<float>`. There is no loss in generality and it simplifies code to do this.
- Color spaces are important when determining the quality of a color sample, and ideally the measurement of two samples can be meaningfully performed by Euclidean distance formula. R'G'B' colors (as computer-generated art and digital photography typically is) are in a nonlinear gamma-corrected space, and slight errors in any one component may be very noticeable. (There is a tremendous rabbit hole here, where you may be tempted to spend a lot of time learning about color spaces. Color encoding and representation is a deep field with mountains of research. Dive in if you must [2].) I found that moving R'G'B' gamma-corrected colors to linear RGB space had the effect of oddly characterizing samples as both closer and farther (at different points along the gamma curve) from each other than they appeared. In short, I found that leaving colors in gamma-corrected space is best. The samples in this article are quantized directly from your standard, run-of-the-mill R'G'B'.
- An obvious optimization is to reduce the number of input samples to only unique values. However, it is worth mentioning that if you do this, make sure each unique sample also carries a weight that is proportional to the number of samples that it represents from the source data. Otherwise, the palettizer will not be able to know how relatively important each sample is, and in general will provide poor fits for highly redundant data sets.
- I tried starting with M different representative values and merging them down to N . The problem is intractable, though, because M may be huge, and the act of finding

"close" values to merge together is $O(n^2)$. This is why very little research exists using this approach.

- Most of the algorithms that start with more than one entry do so by selecting a random set of N representative values for their palette and nudging them around until good values are found. I saw no clear guidance as to how to select these initial values. Seeing as how the initial selection of values greatly influences the number of iterations required to find an optimal fitting, and that it's quite hard to know if the values are stuck in a local minima or are actually good representations, I abandoned these approaches. In tools, I personally prefer to get near-optimal results (by some criteria) in a deterministic number of iterations.
- Once decided upon the strategy of partitioning clusters until N is reached, I still had issues with cycles in the algorithm. Thinking that it would yield tighter clusters, I recomputed the representative value of each partition, then reassigned every sample to the closest value in the palette. This was fine, except that occasionally a cluster would end up with no samples assigned to it. Naturally, I just deleted the value and continued. Don't. This sometimes causes infinite loops when a set of samples cycles between two or three extreme values. In the end, I found that I'd compute the centroid of a set, then find the closest sample in that set and force it to stay associated with the value. At least this way, there is no possibility of an infinite loop. As a further improvement, at the end of an iteration, any cluster that has only one sample assigned to it changes its value to be exactly that sample's value, to be a perfect fit.
- During the reassignment phase, I also experimented with updating the centroid of clusters dynamically. This sounded like a good idea, but it has negative feedback loops based on the ordering of samples. For example, if values are spread out along the number line and there are two values $A=0$ and $B=1000$, assigning a sample 499 to value A will move its centroid to 249.5. Suddenly, the next sample 501 will be closer to A than B , and further skews the cluster positioning. Order dependency is a terrible way to build robust tools and will give highly variable results, so this kind of adaptive behavior is to be avoided at all costs.

7.4 Algorithm Overview

Okay, here's the view from orbit:

1. Initialize the cluster set.
2. Find the cluster with the largest error and divide it into two clusters.
3. Reassign all samples to the closest cluster.
4. Repeat from Step 2 until all cluster errors are zero or the target of N palette entries is reached.

The mental model I work with is that a palette is a bunch of points in space, and each original sample gets mapped to one of those points in space. These are little clouds, or clusters, of samples surrounding each representative value in the palette. Here's a simplified way of writing this in C++:

```
typedef CVector3      RepValue;
typedef CVector3      Sample;
typedef std::vector<Sample>    Samples;
typedef std::map<RepValue, Samples> Clusters;
typedef std::map<Sample, int>    Frequency;
typedef std::vector<float>      AxisWeights;

Clusters    clusters;
Samples     origSamples;
Frequency   sampleFreq;
AxisWeights weights;
```

Step 1 — Initializing the Cluster Set

Reducing the set of samples to a unique set should be done as a preconditioning step for performance reasons. A consequence of doing so is that importance

calculations will need to consider a secondary field that describes the population that each sample represents and is carried through centroid calculations as a weighting factor. Reduction is not complicated, but it is tedious and not required for this algorithm to work correctly. For testing purposes, assign a frequency for each sample in `Frequency` to 1.0.

Initializing the cluster set is accomplished by computing a representative value, `RepValue`, which is the centroid of a set of samples, then populating the `Clusters` map with a single entry having all the samples assigned to it. This is done as follows:

```
RepValue const repValue = ComputeCentroid(origSamples, sampleFreq);

clusters[repValue] = origSamples;
```

Computing a single representative value from many samples is trivial if and only if the numerical space of the samples is linear. For example, colors in a perceptually uniform space can be computed as a simple centroid. Note that since in-put samples x_i have been reduced to only unique entries, a weight factor w_{xi} proportional to its original frequency must be applied whenever considering a sample for computing its contribution to centroid C :

$$C = \frac{\sum_{i=1}^n x_i w_{xi}}{\sum_{i=1}^n w_{xj}}$$

Listing 7.1: Computing the centroid for a set of samples looks complicated because we allow for a reduced set of samples with associated frequencies. It's really just an average.

```
RepValue ComputeCentroid(const Samples& x, const Frequency& w)
{
    int totalFrequency = 0;
    RepValue repValue(0, 0, 0);
    for (size_t samp = 0; samp < x.size(); samp++)
```

```
{
    int const sampleFrequency = w[x[samp]];
    repValue += x[samp] * sampleFrequency;
    totalFrequency += sampleFrequency;
}

repValue /= (float) totalFrequency;
return (repValue);
}
```

Step 2 — Subdivide the Worst Fitting Cluster

Determining the cluster to split is relatively easy, once you have defined the error metric. Iterate over all clusters and compute the sum of errors between the representative value and all its assigned samples, again weighted by the sample's original frequency in the source data. The cluster with the greatest computed error is selected for partitioning. It should be noted that `FindWorstCluster` is the naive implementation, which recomputes the error even for clusters that have not changed. This implementation is simple for clarity's sake, not for performance.

Listing 7.2: This computes the error for each cluster's samples relative to the centroid chosen for it. The cluster with the greatest error is returned for subdivision.

```
RepValue FindWorstCluster(const Clusters& clusters,
    const Frequency& freq, const AxisWeights& aw)
{
    RepValue worstValue(FLT_MAX, FLT_MAX, FLT_MAX); // nonsense value
    float worstError = 0.0F;

    for (Clusters::const_iterator cluster = clusters.begin();
        cluster != clusters.end(); ++cluster)
```

```
{
    const float err = ComputeError(cluster->first,
        cluster->second, freq, aw);
    if (err > worstError)
    {
        worstError = err;
        worstValue = cluster->first;    // remember the worst fit,
        // so we can split it
    }
}

return (worstValue);
}
```

Measuring Error in a Cluster

Computing the quantization error between a sample and its representative value v can be done in a couple of ways if your data forms a vector field. RGB colors and (x, y, z) points are vectors in space that form a metric space, meaning we can use simple distance formulas. You have a choice of simple linear error or squared error. Linear error allows for total deviation ε to be limited, without regard to how well distributed the errors are across the axes of the sample. It seems too lenient, but is fast to compute.

$$\varepsilon = \sum_{i=1}^n |v - x_i| w_{xj}$$

Interestingly, 4-component RGBA colors do not effectively act as a vector field during rendering because a linear change in the alpha may cause unpredictable, nonlinear changes to the final color due to blending. Alpha is not linearly independent of RGB, in other words. That means simple Euclidean distances lose their meaning for the alpha component. The best way I have found to deal with this is to give each scalar

component a weight that is applied during the squared error analysis.

Even so, this technique is helpful because it affords greater control over the fitting of data. Squared error strongly limits the deviation per axis, so that you can control the importance of each scalar within a sample by weighting the deviation on each individual with a factor s_i . This can be used to more heavily weight the Y' luma component rather than chrominance (CrCb), or to improve the z component of normals at the expense of x and y . As you can see, squared error is a little more expensive to compute as well:

$$\varepsilon = \sum_{i=1}^n \left[\sum_{j=0}^2 (v_j - x_{ij})^2 s_j^2 \right] w_{xj}$$

(Here, x_{ij} is the component j of the sample i .)

Listing 7.3: This function computes a squared error between the representative value of a cluster and all of its samples, with a weight per component.

```
float ComputeError(const RepValue& v, const Samples& x,
                  const Frequency& w, const AxisWeights& s)
{
    float err = 0.0F;
    for (size_t samp = 0; samp < x.size(); samp++)
    {
        // compute the error of a single sample
        float squaredErr = 0.0F;
        for (size_t j = 0; j < 3; j++)
        {
            const float linearErr = (v[j] - x[samp][j]) * s[j];
            squaredErr += linearErr * linearErr;
        }
    }
}
```

```
    }  
  
    const int sampleFrequency = w[x[samp]];  
    err += squaredErr * sampleFrequency;  
}  
  
return (err);  
}
```

Splitting a Cluster in Twain

At this point, we've identified the worst-fitting cluster. How do we go about trying to split it? Ideally, the end result will be two clusters with roughly half the samples from the original cluster in each new cluster. Also, we'd like to know that we're splitting the cluster along its longest axis, so that when we divide it in half, it'll be bisecting the longest line we can draw through the data points. This should separate the samples naturally into two groups that are farthest apart from each other.

First, we need a way to determine the principle axis of a set of vectors. Principle component analysis is the class of mathematical tools we're interested in. If we were searching for a fully orthogonal set of axes, there are methods we could implement like eigenvalue decomposition or singular value decomposition. However, this is more work than necessary, since we're never looking for more than the primary axis of a cluster. The primary axis of a set of points is the largest eigenvector of its covariance matrix, which can be easily produced using the power method. This is basically done by continually multiplying a vector against a matrix, normalizing it, and repeating the process until it stops changing direction. The result is the dominant axis of the data in the matrix.

Listing 7.4: This computes a covariance matrix for the samples in this cluster.

```
// Power Method for eigenvectors is taken from
void SplitCluster(const RepValue& v, const Samples& x,
                  const Frequency& w, const AxisWeights& s, Clusters& clusters)
{
    // compute cluster center based on the samples and their frequency
    RepValue center = ComputeCentroid(x, w);

    // count how many pixels this cluster represents
    uint numSamplesRepresented = 0;
    for (size_t i = 0; i < x.size(); i++)
        numSamplesRepresented += w[x[i]];

    // compute covariance matrix of samples
    float covMatrix[3][3] = {{0,0,0}, {0,0,0}, {0,0,0}};
    for (size_t outer = 0; outer < 3; outer++) // rows
    {
        for (size_t inner = 0; inner < 3; inner++) // cols
        {
            // only compute the upper part and diagonals, simply reflect
            // to the bottom part since the covariance matrix is symmetric
            if (inner >= outer)
            {
                // compute the covariance of each channel relative to each
                // other, across all samples.
                float deltaSum = 0.0F;
                for (size_t loop = 0; loop < x.size(); loop++)
                    // weight by number of pixels this represents
                    deltaSum += (x[loop][inner] - center[inner]) *
                                (x[loop][outer] - center[outer]) * w[x[loop]];
            }
        }
    }
}
```

```
    const float covariance = deltaSum / numSamplesRepresented;
    covMatrix[inner][outer] = covariance;
    covMatrix[outer][inner] = covariance; // symmetry
  }
}
```

The above code computes a covariance matrix, which is required to perform the power method for producing the first eigenvector of the data set. There are situations where the power method will not converge—most specifically when the initial eigenvector guess is orthogonal to the real dominant axis. In practice, this doesn't seem to be a very common problem, and if it bothers you, change it to a better method. The appeal in using the power method is that it's simple to implement and understand, and it works pretty well [3].

Listing 7.5: This uses the power method with up to 10 iterations to determine the dominant axis of the cluster.

```
// Power Method for producing the first eigenVector
float eigenVector[3] = {1.0F, 1.0F, 1.0F};
float tempVector[3];
float lastScalar = 0.0F;

// generally converges in ~3 iterations
for (size_t iteration = 0; iteration < 10; iteration++)
{
    // vector-matrix multiply the eigenVector into tempVector
    for (size_t multiI = 0; multiI < 3; multiI++)
    {
        // store this into the temp vector until the multiply is done
        tempVector[multiI] = 0;
```



```
    for (size_t multiJ = 0; multiJ < 3; multiJ++)
        tempVector[multiI] +=
            eigenVector[multiJ] * covMatrix[multiI][multiJ];
}

// normalize the eigenvector (which is not the same as vector
// normalization - a normal eigenvector has a max component of 1)
float maxComponent = 0.0F;

// find the maximum component in the new eigenVector
for (size_t i = 0; i < 3; i++)
    if (fabsf(tempVector[i]) > maxComponent)
        maxComponent = fabsf(tempVector[i]);

// perform the normalization and store into eigenVector
for (size_t i = 0; i < 3; i++)
    eigenVector[i] = tempVector[i] / maxComponent;

// figure out if we've converged or not
const float absoluteRelativeError =
    fabsf((lastScalar - 1.0F / maxComponent) * maxComponent);
if (absoluteRelativeError < 0.001F)
    break; // if our direction has not changed, stop iterating

// move on to another iteration, and remember what the
// eigenvalue was last iteration
lastScalar = 1.0F / maxComponent;
}
```

Once the dominant axis has been determined, project all samples onto this axis and split the set into samples on one side or the other of the plane perpendicular to the dominant axis. In other words, assign all samples based on which side of the midpoint

along the primary axis each sample falls. I suppose we could try using a more sophisticated binning method here, but that would assume we have a realistic representative value in each cluster worth measuring error against. We don't, and the selection of such a sample is NP-hard—the problem of selecting a representative value for a cloud of samples is essentially palettization! At this point, all that matters is that the clusters are roughly equal in size and split along the dominant axis. These new entries are placeholders—samples will be globally repositioned, and new representative values will be computed shortly.

Listing 7.6: This code splits the cluster at the mid-section along the dominant axis, binning half the samples into each new cluster.

```
// Taking any sample, subtract the centroid to get a relative vector
// from the center of the data. Then, take the dot with respect to
// the eigenvector. This results in a value that indicates how far
// along the dominant axis the point is.

Samples  positiveSamples, negativeSamples;

for (size_t i = 0; i < x.size(); i++)
{
    // Since the dominant axis fits the largest range of samples,
    // and the centroid is generally in the center of the samples,
    // splitting there makes sense as a first step.
    float dotProduct = 0.0F;
    for (size_t j = 0; j < 3; j++)
        dotProduct += (x[i][j] - center[j]) * eigenVector[j];

    // Let's bin them out to positive and negative lobes.
    if (dotProduct > 0.0F) positiveSamples.push_back(x[i]);
}
```

```
    else negativeSamples.push_back(x[i]);  
}  
  
RepValue palPositive, palNegative;  
  
// compute the centers of the two new lobes  
RepValue palPositive = ComputeCentroid(positiveSamples, w);  
RepValue palNegative = ComputeCentroid(negativeSamples, w);  
  
// stick both new palette entries into the clusters map  
clusters[palPositive] = positiveSamples;  
clusters[palNegative] = negativeSamples;  
  
// make sure we get rid of the cluster that we've just split, too  
clusters.remove(v);  
}
```

So, there you have it! The above code will partition a set of data into two pieces very well. Still, refinement is required to get a good global fitting to the data. Also, be aware that the data structures shown are for educational purposes only—you can do far better than $O(\log N)$ lookups everywhere.

Step 3 — Reassigning All Samples

Now that we have two smaller clusters where one large, poorly fitted cluster used to be, there is no guarantee that every sample in these two clusters are closest to their centroids. Mathematically speaking, they will be closest to one of the two centroids. Further, samples that previously belonged to other clusters may be attracted to these shiny new centroids because they're a closer fit than their current one. Global sample allocation refinement reduces the measurable sample error, yielding a better palette. Also, future iterations will measure the quality of clusters based on how "well-liked"

they are by their samples. A cluster partitioner that never reallocated samples would end up splitting the wrong cluster eventually, and overall would not fit the data well.

So, first, we clear out all the samples from their clusters. Then, to avoid infinite loops, the first thing we do is iterate over all the empty clusters and find the closest unallocated sample (using the error metric function) and assign it to the cluster. This associates something with every palette entry, so we don't end up with unused spaces in the palette.

Second, we iterate over all the samples and bin them out one-by-one to the closest cluster, using the error metric devised above. At the end, we recompute the centroid of each cluster so that the representative value is ideal for the samples it represents. This process of reassignment could be repeated if desired, as it should be quite stable if the algorithm is implemented correctly. If samples migrate to a couple of clusters, it probably indicates a slightly imbalanced error metric.

Step 4 — Termination

Whenever the cluster count reaches the desired number of palette entries, stop iterating. A final reassignment is effectively the final palettization of the data. In case there are fewer unique samples than palette entries, the error metric should return zero because there is a palette entry for every possible input. Another simple check is that the cluster count did not change over the last iteration. In any of these cases, the algorithm has completed, and the palette and palettized data can be written out.

7.5 Future Work

Error diffusion is an interesting technique that drastically helps heavily compressed images. While definitely not appropriate for non-image data sets, when using error diffusion, one might tweak the algorithm to intentionally reduce to $2N$

colors in the palette, then start attempting to find pairs of colors in the palette that interpolate at 25%, 50%, and 75% to other colors in the palette. Those colors that can easily be interpolated from existing palette entries could be removed from the palette easily and induce relatively little error. Some care would need to be taken to ensure that samples are kept with preference for those that appear most frequently in the source image.

It is plausible that converting the colors to a more meaningful color space, one that interprets the color channels differently by separating important data into one channel, and less important data into other channels, the quantizer could be guided to prefer accuracy on one axis more than others. One such specific color space is $Y'CbCr$. For instance, the chrominance values Cb and Cr can be given lower weighting than the luma value Y' , which is more perceptually relevant. This should help the overall contrast and brightness control of the final image, at some expense to color range. In theory, it sounds like a good idea and is a simple convolution applied to the input data.

7.6 Results

Figures 7.2 and 7.3 illustrate the differences between palettization using Photoshop and palettization using the technique presented in this gem.

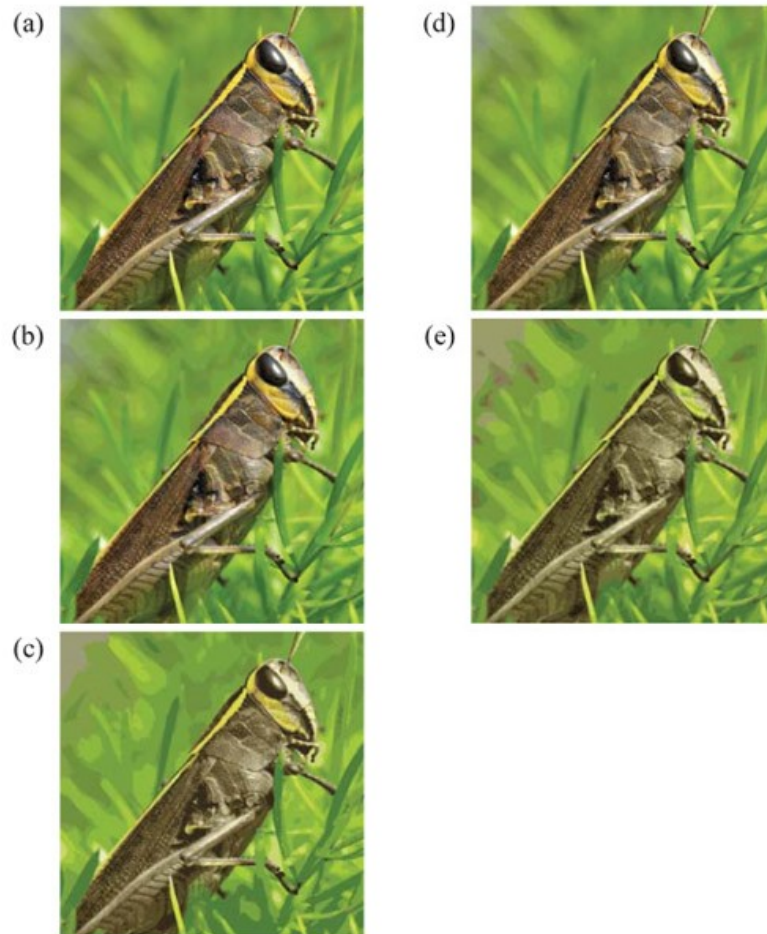


Figure 7.2: (See also Color Plates.) (a) Grasshopper source image using 136,945 colors. (b) Photoshop, 256 colors. (c) Photoshop, 16 colors. (d) WBP, 256 colors. (e) WBP, 16 colors. (© Picstudio/Dreamstime.com)

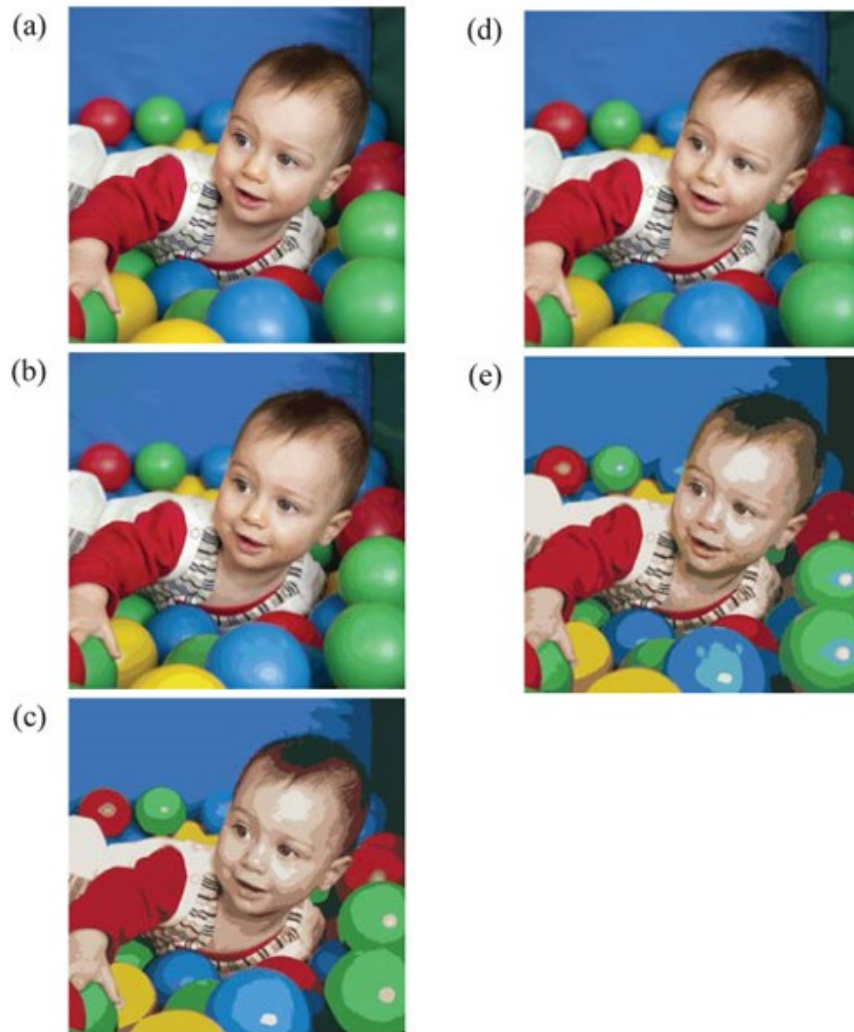


Figure 7.3: (See also Color Plates.) (a) Child source image using 112,024 colors. (b) Photoshop, 256 colors. (c) Photoshop, 16 colors. (d) WBP, 256 colors. (e) WBP, 16 colors. (© Pavla Zakova/Dreamstime.com)

References

[1] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. "NP-hardness of Euclidean sum-of-squares clustering". *Machine Learning*, Volume 75, Number 2 (May 2009), pp. 245–248.

[2] Charles Poynton. "Color FAQ". <http://www.poynton.com/ColorFAQ.html>

[3] E. Garcia. "Power Method (Vector Iteration)". <http://www.miislita.com/information-retrieval-tutorial/matrix-tutorial-3-eigenvalues-eigenvectors.html#power-method>

8

3D Stereoscopic Rendering: An Overview of Implementation Issues

Anders Hast

UPPMAX, Uppsala University

Overview

In recent years, there has been an increasing interest in the field of 3D display technologies from the entertainment industry. Today, the movie industry is moving in at a wide front as thousands of 3D stereoscopic movie theaters are being installed worldwide and movie production companies produce their films also in 3D. In fact, a new job title has emerged, the *stereoscopist*, in charge of making sure that the scenes can be viewed without problems by the audience. Some of the things the stereoscopist must deal with are explained in this gem. At the same time, many products for the home audience have been developed at an affordable price range and sold to a growing number of customers, including stereo capable TV sets and computer screens. This gives the games industry a user base that will be familiar with watching 3D stereoscopic content and who, in the future, might also be expecting their favorite games to be released in stereoscopic 3D. This gem deals with the far most common type of stereoscopic display, the *plano-stereoscopic* display. In contrast to other types of stereoscopic displays, these are displays that work with two planar surfaces in order to

achieve the impression of depth. We base the discussion around the importance of designing the content to fit for a stereoscopic display and the different kinds of viewing conditions that must be considered. Moreover, we discuss the mathematics that help us compute these viewing conditions in order to be able to view the content without any problems. And finally, we provide an overview of different types of display techniques.

We start by briefly familiarizing the reader with the field of stereoscopy and different depth cues, covering some implementation details. We then iterate over some issues that can arise when integrating stereoscopic display support into a game engine. Even though most terms are explained herein, it might be valuable for the reader to use the online glossary provided by [6].

8.1 Mechanisms of Plano-Stereoscopic Viewing

The concept behind plano-stereoscopic displays can be seen quite simply as the creation of two planar views of the game, one for the left eye and one for the right. Then, it is important to make sure that each eye sees only the view intended for that eye. The processes involved can be seen as coding and decoding processes. In scientific visualization, one says that stereoscopic images are aimed to help the viewer form a 3D mental image of the data set. In video games, on the other hand, the aim is to give the player a richer visual experience.

In order to create the sense of depth in a normal rendered game (i.e., a monoscopic rendering), monocular depth cues are used in contrast to the binocular depth cues discussed later. Some examples of *monocular* depth cues are the following:

- *Occlusion* occurs when objects closer to the viewer occlude objects that are further away, and this is handled by the depth buffer algorithm.
- *Parallax* is an effect caused by the motion of the observer, and it creates the illusion that objects close to the observer are moving by faster than objects further away. For

instance, to someone looking out the window of a moving train, trees in the foreground appear to move by much faster than a distant hillside.

- The *size* of an object varies depending on the distance from the viewer due to the perspective projection, where parallel lines converge at the horizon.
- *Texture detail levels* provide information about the distance to an object.
- *Atmospheric effects* such as scattering or haze make objects appear more gray in relation to the distance between them and the viewer.
- *Shading and shadows* tell us about curvature and inter-object relationships.
- The *proximity to the horizon* also provides a depth cue since we know that the horizon is far away, and objects close to the line of the horizon are thus perceived as being far away.

What a stereoscopic display adds to these cues are the three *binocular* depth cues known as *accommodation*, *convergence*, and *retinal disparity*. Convergence occurs when we focus our view at an object in real life by rotating our eyes so that their lines of sight intersect at the point of interest. At the same time, we apply pressure to the lenses in the eyes in order to focus, and this is called accommodation. Under normal natural viewing conditions, both accommodation and convergence correspond and are habitual, but can be voluntarily put out of function by crossing the eyes. The third cue is retinal disparity, and this pertains to the fact that we have two retinal images that fall on different points of the two retinas. These are then merged by the brain and perceived as a single image.

The virtual space that we define is divided by the screen into two regions called *view space* and *screen space*. The volume between the viewer and the display is the view space and the volume behind the display is called screen space. If we look at a point lying at the same depth as the display surface onto which it projects, as shown in the left image of Figure 8.1, then the homologous points on the display have zero parallax

because they have no lateral displacement. The homologous points are identical features in the stereo pair; thus, the same point in space is located on different places in the left and right stereo image for nonzero parallax. Points that are lying in either view space or in screen space have a lateral displacement and are then said to have either negative or positive parallax. All three cases are shown in Figure 8.1. Converging at a point behind the display surface causes the homologous points on the display to have positive parallax. This point is said to be in screen space. Similarly, converging at a point in front of the display surface causes the homologous points on the display to have negative parallax, and this point is said to be in view space.

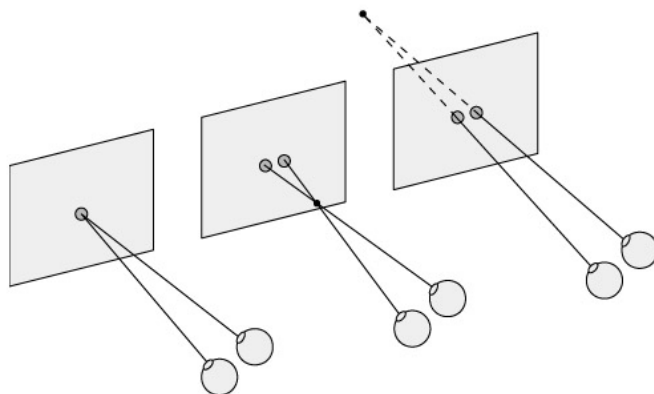


Figure 8.1: Three different types of parallax that can occur, from left to right— zero parallax, negative parallax, and positive parallax.

Converging the eye's axes upon a virtual point at distance D_c supports the fusion of the parallax image and stereopsis as shown in Figure 8.2, where stereopsis is the mental and psychological process in visual perception leading to the sensation of depth from two slightly different projections of the world onto each eye [1]. Keeping the visual structure on-screen in focus requires accommodation at screen distance D_s . Usually, accommodation is dominant, and for unaided viewing, we see one planar double image

in focus. With some training (crossing the eyes) we can converge at D_c and see a fused 3D image out of focus. Stereographic devices such as stereo glasses greatly support image fusion and stereopsis at the cost of suppressed accommodation.

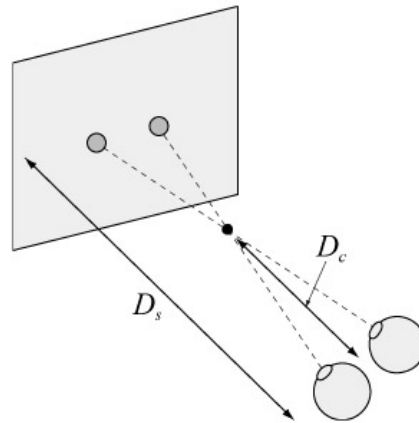


Figure 8.2: Convergence and accommodation in plano-stereoscopic displays.

Scale Considerations for 3D Stereo Images

To be able to create comfortable stereo images, we need to calculate the correct perspective for the given setup and make sure that we stay within those limits [8]. Some new situations arise from using a stereoscopic display. Mainly, scale considerations are important when modeling the virtual view volume and creating the actual stereo pair.

When converging on objects at some certain distance, a point at a different distance in the scene appears at the retina with some lateral disparity, independent of convergence, as shown in Figure 8.3. The inter-pupillary distance (*IPD*) is the distance between the eyes measured from the center of the pupils in each eye. The retinal disparity is, according to Kalawsky [12], computed as the difference of the two angles shown in the figure:

- $disp = \theta_1 - \theta_2$.

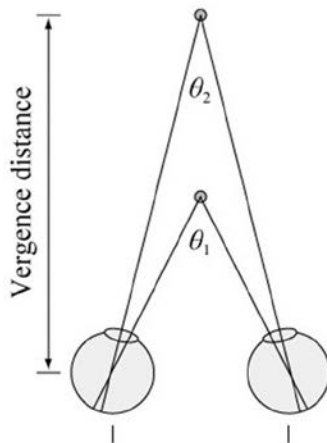


Figure 8.3: The two angles used to compute the retinal disparity.

A retinal disparity of more than 10° causes diplopia (double vision) and should, of course, be avoided at all times.

The projection in stereoscopic displays does not scale linearly. The projection of the two points that we fuse in our brain is very dependent on how far away we sit from the screen and how big the screen is. As this point is projected onto our eyes, the distance between our eyes is also a big impact factor. An average person has an eye separation around 6.5 cm. In a more uncommon case, we can find people with up to 7.5 cm between their eyes, and the largest audience or user group, youngsters, can have as few as 4.5 cm between their eyes. And it should be remembered that there is a great variation among people, particularly on different continents [4]. Some good advice would probably be for the IPD to be an adjustable variable set by the player in order to give a comfortable stereo depth.

The physical limit on our depth perception is around 200 yards. This comes from

the fact that beyond this point, we don't get any convergence information at all since our line of sight is more or less parallel. This important fact must be considered when we create our virtual space, which we must design before we start to place objects in it.

Another practical issue in stereo graphics is that one should not exceed parallax values of more than 1.5° visual angle in order to not feel uncomfortable [13], as shown in Figure 8.4. The on-screen parallax (*osp*), measured in centimeters for different viewing distances D , is shown in Table 8.1.

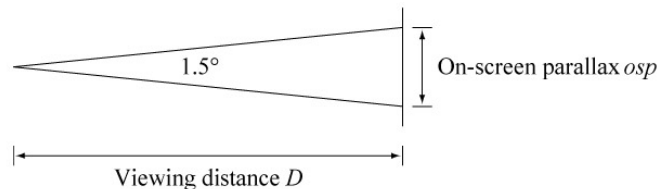


Figure 8.4: Parallax values greater than 1.5° visual angle should not be exceeded.

Table 8.1: Practical examples for on-screen parallax values.

D (cm)	On-screen-parallax (cm)
50	1.31
75	1.96
100	2.62
200	5.24
300	7.86
400	10.47

Let us now look at some practical examples of how the virtual space depth is limited due to the distance to the observer and the *osp* for a negative parallax situation, as shown in Figure 8.5, where d is the virtual spatial depth. We have

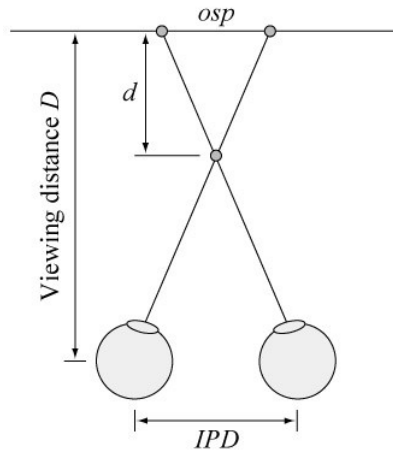


Figure 8.5: Negative parallax.

and solving for d gives us

$$d = \frac{D}{IPD/osp - 1}$$

Similarly, we can compute how the virtual space depth d is limited due to the distance D to the observer and the osp for a situation with positive parallax as shown in Figure 8.6. Tables 8.2 and 8.3 show some example values for the cases of negative and positive parallax.

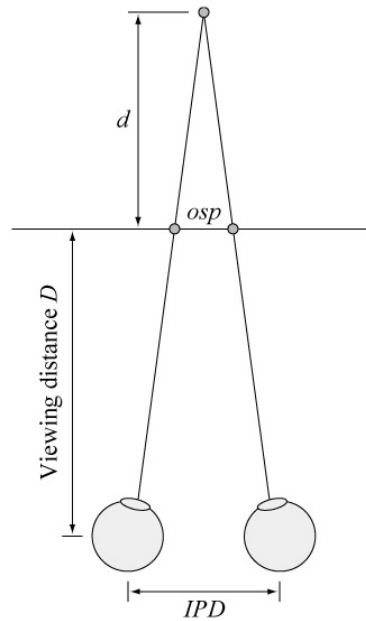


Figure 8.6: Positive parallax.

Table 8.2: Example for $IPD=6.5$ cm, for negative parallax.

D (cm)	osp (cm)	d	d/D
50	1.31	8.38	0.17
75	1.96	17.40	0.23
100	2.62	28.72	0.29
200	5.24	89.24	0.45
300	7.86	164.17	0.55
400	10.47	246.83	0.62

Table 8.3: Example for IPD=6.5 cm, for positive parallax.

D (cm)	osp (cm)	d	d/D
50	1.31	12.61	0.25
75	1.96	32.47	0.43
100	2.62	67.47	0.67
200	5.24	829.45	4.15
300	5.76	1714.7	7.79
400	6.42	18612.4	75.97

Before you go ahead and work out the math for your game, there are some things worth mentioning. First, one should notice that values for maximum allowable disparity in the literature are expressed in different terms such as angular disparity, on-screen parallax, etc. These values sometimes apply only as a rule of thumb. Lipton's values appear to work for many VR applications, but they do not generally apply for all viewing conditions. According to Lipton, compositing stereoscopic images is an art, not a science. In the end what matters is that it looks and feels good!

The Basic Setup

Several propositions have been made on how to setup and render graphics for stereoscopic displays as efficiently as possible. One way is to let the graphics driver handle it. Some of the major graphics vendors have native support for stereoscopic displays in their drivers where it creates the stereo pair. This does require that the game is compatible with stereoscopic display in the form that the content is conformed for these kinds of systems. And this limits your game to be used only for these specific graphics card vendors in order to make your game work in stereo. The second, less common alternative in the context of the game industry is to make a graphics command

interceptor [3]. This is an application that pretends to be the graphics driver and intercepts all the graphics sent from an application. This can be stored and then in theory be used to create the two stereo pairs. It is still required that the game create content that is suitable for stereoscopic displays.

Usually, we would use the following simple approach to create the game in stereo.

```
while (user wants to play)
  doGameLogic()
  setupProjectionForLeftEye()
  render()
  setupProjectionForRightEye()
  render()
```

In our game, we must set the virtual cameras to focus on the point of interest, as shown in Figure 8.7, using the previously described math in order to ensure that the parallax do not exceed the values for the *allowable* virtual space depth.

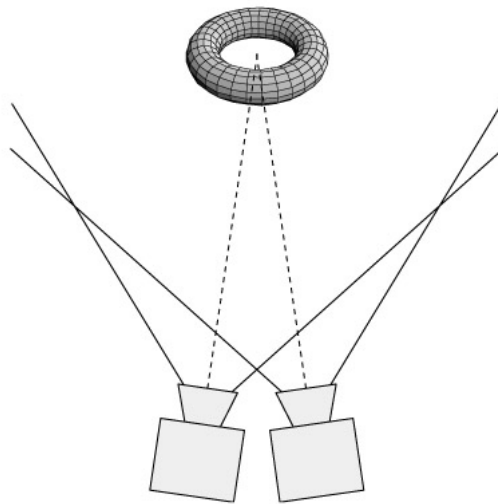


Figure 8.7: Camera setup for the 3D stereoscopic game.

8.2 Stereo Techniques

Today, there are three different popular techniques of viewing graphics in stereo on the market: anaglyph stereo (a.k.a. red/green stereo) [2], temporal multiplexing, and polarization. We give a brief overview of these techniques and mention some pros and cons as well as examine the coding and decoding process for each of them. An important concept discussed is *ghosting* [14], which should be avoided as much as possible. Ghosting means that one eye sees some of the content meant for the other eye.

Anaglyph Stereo

Anaglyph stereo is the simplest and cheapest way of delivering stereoscopic game content to the players. Here, the left and right views are separated using wavelength separation. The left view is simply encoded in the red channel and the right view is encoded with complementary colors such as the green channel or both blue and green (cyan). The player needs to wear the well-known red/green glasses in order to separate the two images so that each eye sees only the image meant for it. Hence, the color of the lenses in the glasses corresponds to the color channels that encode the picture, and this is the reason for its name. The source is encoded into one image, which means that this method can be used for a lot of different media, even in printed form.

The benefit of using this technique for games is that it does not require any special display system, just a pair of cheap red/green glasses in order to be able to view the effect. You've probably gotten a pair of cardboard glasses with acrylic lenses for free when you bought your favorite comic magazine that had a special 3D centerfold issue.

One pitfall using this technique is that the player must calibrate the screen so that the color matches the filtered glasses. Otherwise, the player sees a lot of ghosting with a bad quality stereo effect. There is also a noticeable loss of color, as a lot of the color information is being filtered out.

A more sophisticated version, sometimes called the super anaglyph technique, uses spectral multiplexing, more often referred to as Interference Light Technology (INFITEC) [9]. Spectral multiplexing works by dividing the visual spectra into six narrow bands, two bands for each of the primary colors red, green, and blue. These bands are then separated by filters and divided so that one band of each color reaches each eye. That is, half of the red spectra reaches the left eye, and the other half reaches the right eye, and so on. This system requires that two such filters of different type performing this spectral multiplexing are mounted onto two projectors for the display, as well as a pair of lightweight glasses that the player wears with one type for each eye. Hence, the viewer clearly sees, if he shuts one of his eyes, that the color information reaching the eye is slightly different compared to looking at the display using only the other eye. Nonetheless, the color differences are less than that seen when using the red/green glasses. Finally, the brain merges these two images into one without a problem, and the picture looks as expected.

Temporal Multiplexing

This technique encodes the stereo pair by interleaving them time-wise. Hence, one frame is being shown for the left eye while the right eye is being occluded by an active shutter. Similarly, the next frame is visible to the right eye while the left eye is being occluded. If this procedure is performed fast enough, the player perceives the interleaved images as one continuous stream of images and gets the stereopsis right. This technology cuts the effective frame rate in half, as it is necessary to render twice as many images to get the same update rate. The occluding is performed by a pair of active shutter glasses, usually a pair of LCD screens that is synchronized to the display. While the display system shows a new frame, it sends a signal to the glasses to make the shuttering. It is clear that this technique requires glasses that cost a lot more than the simple red/green glasses. Furthermore, it cannot be used for printed media since, clearly,

the display system has to have two sources for the output.

Polarized Light

Image separation can also be achieved by encoding each image using polarization. The two images are superimposed onto the screen by the display system through a pair of orthogonal polarizing filters. The viewer also wears a pair of glasses with corresponding filters, which are relatively cheap compared to the active shutter glasses. The polarization directions in the glasses correspond to those on the source. Thus, no extra hardware is needed in the glasses, but once again it is necessary to have two sources of light, and this technique cannot be used for printed media. The viewer must not tilt his head when using this technique, as it results in severe ghosting. Alternatively, circular polarization can be used in an attempt to avoid this problem. In all cases, polarized filters dim the brightness of the source because the original content is filtered.

Summary

All three techniques previously mentioned (with the exception of the INFITEC variation of anaglyph stereo) have become mainstream for playing stereo games. A pair of anaglyph glasses has a very low price and does not require the player to invest in any additional hardware to enjoy stereo games, but it produces the worst color representation of the three mentioned techniques. However, the frame rate does not have to be increased as for temporal multiplexing. Temporal multiplexing requires that the player acquires the actual glasses and also has a source that can preferably emit 120 Hz, since the actual frame rate is cut in half with this technique. The current trend is that more of this type of display and TV are coming out. The polarized light technique is realized either as an actively polarized display or by having two projectors for the really luxurious players. Regardless, the glasses are passive and thus generally cheaper.

8.3 Design Considerations for 3D Scenes

What really differs when creating content for a stereoscopic game engine compared to a monoscopic one? There are some approximations that can look very bad or very flat when ported to a stereoscopic display. We discuss some of them in this section. It is also important to avoid some situations that can be unpleasant for the player [10].

Culling can be a problem in the sense that we have two frusta to clip against. Clipping against the monoscopic frustum would yield erroneous results, as this is smaller than the combination of the left and right frusta. This problem can be handled in at least two ways, either clip against the joint frusta from left and right at once or clip separately against both frusta.

Stereoscopic rendering in its naive form, by rendering the scene twice from two different perspectives, occurs at around twice the cost. Some effects do not need to be calculated twice, for example, those that are view point independent. Some of the shortcuts that are normally made in monoscopic rendering do not work in stereoscopic rendering. If these visual effects are not gameplay critical, like glows around objects, they can be turned off or replaced by similar object-based effects.

The offscreen buffers only need to be duplicated if they have some view-dependent effect. This applies to shadow mapping and to those variants where the shadow map is based on the setup of the view frustum. Many of the often used screen-space post processing effects do not look good in stereo. One thing to be careful with is high dynamic range (HDR) rendering. The contrast between left and right view is very important, and if we apply tone mapping separately to left and right eye, we can introduce a shift in contrast that induces strain to the player.

Billboarding is another commonly used effect that does not port well to

stereoscopic displays. Billboards, which are 2D objects that always face the viewer, look under certain conditions like two planar objects and not like a volumetric object as they were intended. The same applies to impostors—they must be made eye-dependent to not look completely flat on a stereoscopic display. This also incurs a performance penalty since twice as many impostors must be rendered.

Backgrounds must, on a stereoscopic display, be placed at the proper depth so that they appear to be as far away as backgrounds usually are. This is especially important with skyboxes and skydomes, as they otherwise appear to be placed too close to the player as if the sky is part of the ceiling. This also relates to the depth range in your scene. If techniques are used that require different depth ranges, these also destroy the stereoscopic sensation. Different depth ranges imply different depth in stereo, and objects end up in unexpected places in the virtual room, causing them to look deformed.

The same effect can be seen with overlays such as graphical user interfaces (GUIs), which are normally rendered in screen space without depth information. However, in stereo this gives contradicting depth cues because the menu can be drawn over an object that lies in view space, but the occluding cue implies that we should see the object. The GUI should lie behind the object, but as consequence of the screen-space rendering, lies at the wrong depth. One solution to this problem is to always put the GUI closest to the player. Depending on how the virtual room is set up, this solution might shove the GUI into the player's face so it must be done with great caution.

Other effects that traditionally take place in screen space, such as text labels, must also be placed in the game world at the proper depth. This now means that they can also be covered by objects in front of them, which might change how the actual game plays. In combination with GUI and text labels, we often also have some kind of representation of input devices. The standard mouse cursor rendered by the operating system can often cause anomalies in different ways. If we are rendering the content to a side-by-side buffer, which is a common technique to render 3D content, we only see the

cursor in one eye since both eyes share the same buffer. Seeing an object with only one eye that should be seen by both eyes is really annoying and should always be avoided!

Thus, we have the same problems as with the GUI when the pointer is rendered in screen space. It is therefore advisable to create a software cursor that is placed as an object with depth in the world.

It is really tiring for the eyes, and the viewer even loses the 3D effect for a short while, if the focus from one scene to another changes dramatically. If an object appears very close to the viewer in one scene, then it can be dragged back a bit before the scene changes, or rather, the focus needs to be pulled back to the place of focus for the next scene. This is something that traditional games usually never bother with, but will probably be a great challenge for stereo game developers. A similar problem occurs for stereo films with subtitles, as they will definitely make the viewer tired since the focus needs to be changed repeatedly. The viewer will probably end up concentrating on the film without even bothering to read the subtitles.

Situations with contradicting depth cues in relation to occlusion can also arise when objects move off the screen in view space. Now, the depth cue tells the player that the object is in front of the screen, but when it comes close to the right or left edge of the screen it disappears behind it. To make this so-called edge conflict a bit less apparent, we can move the object slowly off screen or introduce virtual borders on the screen. These would be two guard bands lying in depth closest to the viewer to prevent the contradicting depth cues. This solution is known as floating windows and was used as early as the 1950s [15]. It has so far not been used for computer games, but was recently used for the Pixar film *Up*.

We also see a need for a greater lower bound of the accepted frame rate in a game when migrating to stereo. People tend to experience jerkiness if the frame rate drops below 60 FPS when viewing it in stereo, a far greater number than with monoscopic

displays. Depending on what type of viewing device the player uses, we can also experience problems with frame-sequential delivery of stereographical content.

8.4 Outlook

Using stereoscopic displays is a great way to increase the gameplay value in the form of immersion. This is still somewhat of an unexplored field even though stereoscopic displays have been used for a longer period of time in other fields such as scientific visualization. The film industry has more experience than the game industry with stereoscopic displays, and we are now starting to see more and more movies created for these types of displays. When will the game industry follow in those footsteps and release more games that are directly catered for stereo enabled devices? So far, the majority of the games that are stereo capable are ports from monoscopic games. The step from monoscopic to stereoscopic also gives us game developers a new dimension to create more interesting games design-wise. After all, 3D stereoscopy is regarded by the film makers as yet another important storytelling technique. The future will hold many new design approaches, both from a pure rendering perspective and also from a gameplay perspective. An interesting approach is to render the scene from one virtual camera position and use the information in the depth buffer to generate a stereo pair using Depth-Image-Based Rendering (DIBR) [7]. A simple example of a unique gameplay experience available only in stereo could be the sniper scope common in many games involving guns. The standard way of handling this is to render a circle in the middle of the screen where the player sees the zoomed-in piece of the world. The stereoscopic version could then, when the player zooms with the scope, black out one eye and the player would also lose the stereopsis, giving more realism and immersion.

The future also holds other types of stereoscopic displays called auto-stereoscopic displays. These displays function without glasses, but must render many more than two

views [11, 5]. Even though the technique has been around for a long time, it still has not been widely accepted because it has some drawbacks such as ghosting. Making sure that your game engine is adopted for plano-stereoscopic displays is a step towards making them work smoothly with auto-stereoscopic displays, and the guidelines presented in this gem will help you to achieve this as a game engine designer.

Acknowledgements

The author wishes to thank Stefan Seipel, Uppsala University, and Martin Ericsson, UPPMAX, Uppsala University, for contributing their material for this gem.

References

- [1] Akiyuki Anzai, Izumi Ohzawa, and Ralph D. Freeman, "Neural Mechanisms for Processing Binocular Information". *Journal of Neurophysiology*, Volume 82, Number 2 (August 1999), pp. 891–908.
- [2] Anaglyph. http://en.wikipedia.org/wiki/Anaglyph_image
- [3] Chromium. <http://chromium.sourceforge.net/>
- [4] Neil A. Dodgson. "Variation and extrema of human interpupillary distance". *Proceedings of SPIE Stereoscopic Displays and Virtual Reality Systems XI*, 2004, pp. 36–46.
- [5] Neil A. Dodgson, J. R. Moore, and S. R. Lang. "Multi-view autostereoscopic 3D display". *IBC (International Broadcasting Convention) 1999*, pp. 497–502.
- [6] International Stereoscopic Union. "International Stereoscopic Union: A Glossary of Stereoscopic Terms". <http://www.stereoscopy.com/isu/glossary-index.html>

- [7] Julien C. Flack, Hugh Sanderson, Steven I. Pegg, and Simon Kwok. "Optimising 3D image quality and performance for stereoscopic game drivers". *Proceedings of SPIE Stereoscopic Displays and Applications XX*, 2009.
- [8] Graham R. Jones, Delman Lee, Nicolas S. Holliman, and David Ezra, "Controlling perceived depth in stereoscopic images", *Proceedings of SPIE Stereoscopic Displays and Virtual Reality Systems VIII*, 2001, pp. 42–53.
- [9] Helmut Jorke and Markus Fritz. "INFITEC—A New Stereoscopic Visualisation Tool by Wavelength Multiplex Imaging". *Journal of Three Dimensional Images*, Volume 19, Number 3 (September, 2005), pp. 50–56.
- [10] Jukka Häkkinen, Monika Pölönen, Jari Takatalo, and Göte Nyman. "Simulator sickness in virtual display gaming: a comparison of stereoscopic and non-stereoscopic situations". *ACM International Conference Proceeding Series*, Volume 159 (2006), pp. 227–230.
- [11] Ken Perlin, Salvatore Paxia, and Joel S. Kollin. "An Autostereoscopic Display", *Proceedings of the 27th Annual Conference on Computer graphics and Interactive Techniques*, 2000, pp. 319–326.
- [12] Roy S. Kalawsky. *The Science of Virtual Reality and Virtual Environments*. Addison-Wesley, 1993.
- [13] Lenny Lipton. *The CrystalEyes Handbook*. StereoGraphics Corporation, 1991.
- [14] Bernard Mendiburu. *3D Movie Making: Stereoscopic Digital Cinema from Script to Screen*. Focal Press, 2009.
- [15] Raymond Spottiswoode and Nigel Spottiswoode. *The Theory of Stereoscopic Transmission and its Application to the Motion Picture*. University of California Press, 1953.

9

A Multithreaded 3D Renderer

Sebastien Schertenleib

Sony Computer Entertainment Europe

Overview

The 3D renderer remains one of the main components in most modern video games. Usually, 3D rendering engines handle both the software and hardware pipelines that are being exposed through 3D graphics APIs such as DirectX, OpenGL, or libgcm. Nowadays, multi-core CPUs are widely available through game consoles and PCs. In order to ensure that the GPU is continuously fed with data to process, it is critical that 3D renderers take full advantage of this new programming scheme by utilizing the available processing cores. The workflow involved in creating a 3D picture on the screen relies on preparing a list of commands that the GPU can interpret and execute.

Different approaches can be taken to decouple the CPU from the GPU. For instance, one common technique consists of using a double buffer or triple buffer scheme where the CPU builds the commands in frame N and where the GPU consumes them in frame $N+1$, as illustrated in Figure 9.1. Alternatively, the GPU can consume the data within the same frame in order to reduce the latency, as shown in Figure 9.2. One potential drawback is that it might be difficult to avoid some GPU stalls early in the frame. On the other hand, the memory footprint is much smaller compared to a double buffering method, and this is particularly important on embedded systems with

restricted amounts of memory.

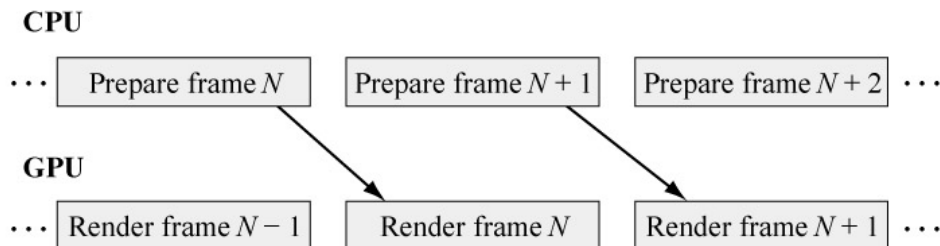


Figure 9.1: In a double-buffering scheme, the GPU consumes the data a frame later than it is generated by the CPU.

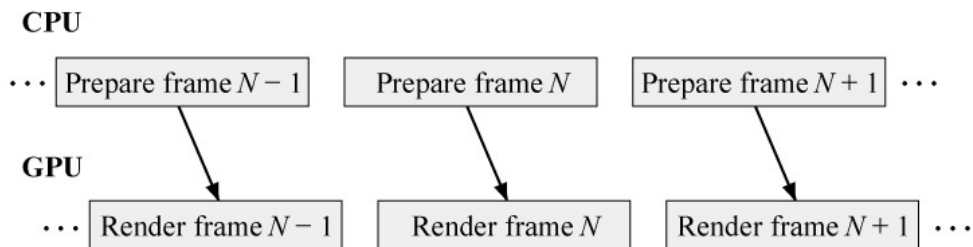


Figure 9.2: In this scheme, the GPU consumes the data soon after it is generated by the CPU.

9.1 The Memory Model

Regardless of how the display lists get created, a renderer might ultimately be restricted by memory access speed, particularly when the graphics commands are generated through a single thread. In recent years, the increasing performance gap between GPU processing power and memory latencies has made it harder to feed the GPU due to expensive data accesses in system memory, as shown in Figure 9.3. A typical frame is subdivided into passes which handle, for instance, rendering shadow maps, rendering the main scene, and rendering fullscreen post-processing effects. A unit of

work during rendering is often referred to as a *batch* and combines a set of render states, shaders, and geometry elements as exemplified by the following listing:

```
//setting up a batch
setRenderStates(...);
bindTextures(...);
setShaders(...);
setShaderConstants(...);
setVertexBuffer(...);
setIndexBuffer(...);
drawCall(...);
```

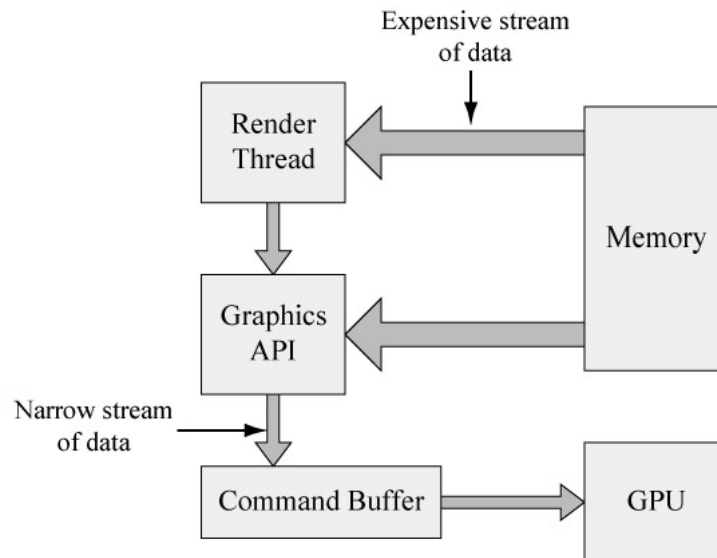


Figure 9.3: Both the render thread and the graphics API are likely to access data in various locations in memory.

Setting up a batch consists mostly of setting the addresses of various resources needed to render an object. In the process of creating batches, the rendering code has

to traverse the scene graph and is likely to access data in various locations throughout main memory. This leads to a large number of cache misses. With inorder CPUs such as the PowerPC chips found in the Xbox 360 and PlayStation 3 game consoles, this could stall the processor for hundreds of cycles each time a cache miss occurs, greatly impacting performance. This problem can be mitigated with out-of-order CPUs because other instructions can potentially be executed while previous instructions are waiting for data to be ready.

One may argue that it might be possible to reorganize the data structures to avoid many of the cache-miss penalties. Indeed, adopting cache-aware or cache-oblivious algorithms [2] helps, especially for the scene graph management, but unfortunately, the graphics library also needs to access and manipulate some data on its own. Large structures such as vertex buffers and index buffers are generally stored in the GPU's local memory and so do not cause a problem, but many types of rendering state, such as shader constants and texture configurations, need to be copied to the command buffer each frame. Therefore, alternative solutions are needed to overcome the cache limitations.

9.2 Building the Display Lists in Parallel

To ensure that the display lists are created in the shortest amount of time and in a way that minimizes memory bandwidth limitations, our solution uses multiple threads, and each is responsible for creating a subset of the rendering commands. However, this is only possible when the 3D graphics library exposes such a level of granularity. Thankfully, this is the case for Xbox 360, PlayStation 3, and DirectX 11 developers. In the PlayStation 3 case, the Synergistic Processing Units (SPUs) of the Cell Broadband Engine, being purely vector processors, excel with geometric processing. This means they can easily perform graphical operations that help offload work from the GPU when

necessary. To create the display lists in parallel, a commonly found paradigm is to have the primary command buffer reference display lists created inside secondary command buffers. Those display lists are created in parallel on different execution units as shown in Figure 9.4. Often, those secondary buffers handle a subset of the frame, and the granularity could be anything from a single draw call to an entire pass.

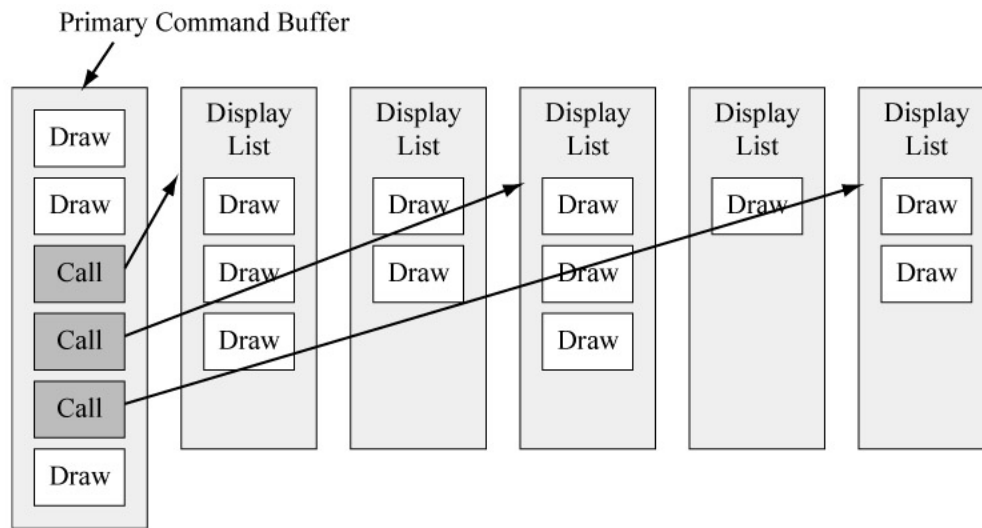


Figure 9.4: The primary command buffer references multiple secondary command buffers, each of which handles a subset of the frame.

By distributing the creation of the draw calls and their graphics states to multiple command buffers in parallel, the overall latency for creating the display lists is considerably reduced, which is particularly useful when using the scheme shown in Figure 9.2. Moreover, this makes the 3D renderer more scalable to various configurations of multi-core architectures and helps generate the thousands of draw calls commonly found in modern video games.

9.3 Parallel Models

3D renderers are strong candidates for parallelization because each draw call can usually be treated as a standalone unit of work. Each of these tasks pairs a chunk of data with some logic to operate on that data as shown in the following code listing.

```
int TaskMain(...)
{
    const int sourceAddr = ...; // source address of data
    const int count = ...;      // number of elements
    const int dataSize = count * sizeof(gfxObject);

    // On some platforms like the PS3,
    // you may have to keep the data local to the executing unit
    gfxObject *buffer = (gfxObject *) Allocate(dataSize);
    DmaGet(buffer, sourceAddr, dataSize);
    DmaWait(...); // barrier to wait for the data

    // let's do some work
    for (int i = 0; i < count; ++i)
    {
        buffer[i]->update();
    }

    // On some platforms like the PS3,
    // you may have to store back the data to system memory
    DmaPut(buffer, sourceAddr, dataSize);
    DmaWait(...); // barrier to wait for the data
}
```

This model provides an efficient parallel paradigm, where each task can be queued and consumed by available processing units, as shown in Figure 9.5. This also avoids

some of the issues of a more standard multithreaded approach, as the tasks provide a more fine-grained subdivision and can cope more easily with uneven computation, while a multithreaded architecture might end up waiting on a particular subsystem to complete its tasks.

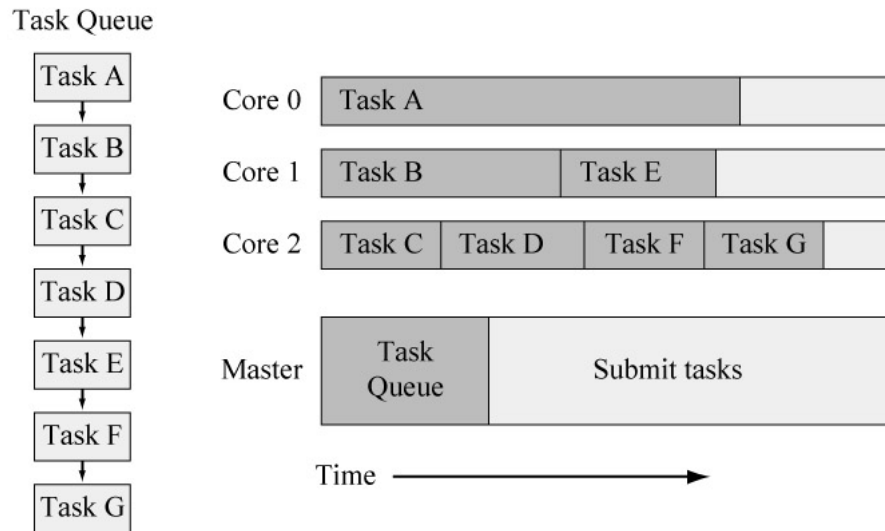


Figure 9.5: Tasks are stored in a queue and are consumed by available processing units.

9.4 Synchronizing the GPU and CPU

Synchronizing the GPU and CPU is more difficult to handle in a multithreaded environment since multiple processing units might potentially want to access the same data. To avoid any inconsistencies and race conditions, there is a need to employ synchronization primitives such as mutexes or atomics. Usually, the GPU can report its progress in a specific memory area. For instance, when a particular command has been finished on the GPU, it can write a specified value, or "report", to a CPU-accessible

location to indicate completion. Depending on the architecture, the CPU can either poll for the report value or receive a system callback of some kind. Table 9.1 presents some possible configurations.

Table 9.1: Synchronizing the GPU and the CPU.

CPU	CPU	Report	Comments
...	WaitReport(22)	0	GPU waits for report to be set
SetReport(22)	WaitReport(22)	22	GPU is now unlocked
...	...	22	
WaitReport(33)			CPU wait for report to be set
WaitReport(33)	SetReport(33)	33	GPU set the report & unlock CPU
...	...	33	
SetReport(12)	SetReport(15)	?	Race condition
WaitReport(33)	WaitReport(22)	?	Deadlock

As with any multithreaded environment, special care is needed to avoid potential race conditions or deadlocks since both the CPU and the GPU can generate and consume data with unpredictable timing patterns.

9.5 Using Additional Processing Resources

In many games, the processing load is not fully balanced among the available processing units. In these cases, it can be worthwhile to move some operations that are ordinarily performed on the main CPU to other units that may have idle time each frame. For instance, less intensive graphics applications can employ GPGPU (general purpose GPU) code to offload physics or AI simulations from the CPU using technologies such as CUDA. On the other hand, games willing to push the boundaries

of real-time 3D graphics might want to use spare CPU cores to perform some graphical operations. If it so happens that those cores provide an interesting ISA (instruction set architecture) with SIMD instructions such as the SPU's on the Cell processor, then it becomes possible to offload the GPU for several kinds of operations such as the following:

- Geometric processing, including procedural algorithms for creating terrain, trees, decals, or subdivision surfaces.
- Physics and particle system updates.
- View frustum object culling or occlusion culling.
- Software rendering, in particular for occlusion queries and post-processing effects.

9.6 Reducing the Pressure on the Memory Bandwidth

The performance of both the CPU and GPU is improving at a very fast pace, but memory speed is not following the same curve, and consequently, memory bandwidth becomes more and more of a significant bottleneck. Therefore, it is important to consider any technique that would help minimize the requirements on the memory systems, even if it means using more CPU or GPU cycles. Some techniques that can be used include packing the shader input and output attributes or using the tessellation units of the GPU when available. On the CPU side, special geometry culling can avoid sending up to 70% of primitives that end up being discarded by the GPU (backfacing, off-screen, zero-size, and degenerate primitives). Another technique is to generate a coarse depth buffer in software to perform occlusion queries instead of relying on the GPU [1], as the latter can involve reading back from video memory, which is often a slow path.

9.7 Performing Graphical Operations in Parallel

So far, we have discussed some techniques that help to improve overall performance, but we can go a step further and allow different processing units to work in parallel to create the final image for a frame. Modern games use multiple render targets within a single frame, and some of them are not accessed simultaneously by the GPU. For instance, it is often possible to access a back buffer with another processing unit such as an SPU to perform some post-processing effects while the GPU starts rendering the next frame. This gives up to a full frame to render the effects, while keeping the same frame rate, but at the expense of an additional frame of latency (see Figure 9.6).

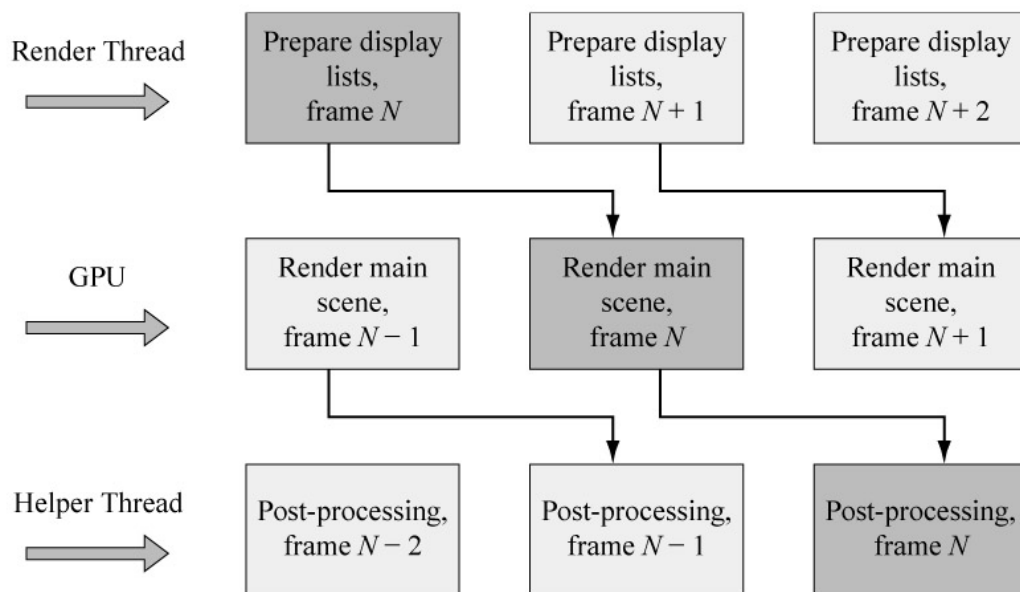


Figure 9.6: While the helper thread computes the post-effects, the GPU starts rendering the next frame.

References

- [1] Johan Andersson. "The Intersection of Game Engines and GPUs: Current & Future". *Graphics Hardware 2008*.
- [2] Sebastien Schertenleib. "An Effective Cache-Oblivious Implementation of the ABT Tree". *Game Programming Gems 5*, Charles River Media, 2005.

10

Camera-Centric Engine Design for Multithreaded Rendering

Colt McAnlis

Blizzard Entertainment

Overview

Modern graphics APIs grant the ability to render in parallel by allowing the creation of drawing commands on separate threads. As such, an advanced engine design must have the ability to scale practically in order to take advantage of increased core count for rendering. In order to accomplish this, an engine must solve the problem of how to properly break up rendering tasks to be computed in parallel, but in order to do that, it must first consider what the rendering subsystem is actually doing.

Modern graphics engines require multiple renderings of the same scene to aid in the overall appearance of the final image. For instance, separate renders of a scene are required to compute shadow mapping, run-time reflections, screen tiling (for antialiasing on some game consoles), imposter generation, and offscreen particles. As such, it makes logical sense that the 'camera' is the coarsest form of scene organization directly tied to the visibility of an environment, given that in order to visualize any of the concepts above, you must first define a view position, view direction, and eventually a render target in which to store the results. As most views of the scene can be rendered

independently of each other, we submit that grouping rendering workload by camera view offers the best rendering workload organization to take advantage of multi-core rendering.

In this gem, we present an engine design that scales to multi-core systems by using the concept of a camera to separate parallel rendering jobs. To accomplish this, we present two concepts. The first is an API-independent method of creating recordable command buffers, a process that enables us to use parallel rendering on any device. The second is an observation of how to distribute the creation of command buffers to separate threads based upon the camera with which it will be used. Using these two simple, yet powerful concepts can enable your engine to take advantage of multiple cores for rendering with the least amount of pain possible.

10.1 Uses of Multi-Core in Video Games

With the boom of multi-core system availability, there's a mad dash to fill extra cores with proper amounts of work to enhance the look of our products. The downside, though, is that for developers on platforms where the hardware configuration can change between two users (or even the same user), decisions must be made about how to accurately scale out visual features and workloads on lower-end processors.

Typically, we run in parallel things such as particles, animations, physics, etc., which can have level-of-detail (LOD) built into them for low-end machines. This concept, however, doesn't directly translate to the act of submitting commands to the graphics API, a process which can easily incur a great deal of performance overhead. APIs such as DirectX 9 and DirectX 10 suffer greatly from this, as it's not uncommon for frequent calls of basic API functions to become a performance burden. This typically causes the rendering phase to delay the processing of other systems in the architecture, which if highly dependent on refresh rate, could cause problems. The cause of this issue

is the fact that we are required to submit commands to the device on the thread that owns the device, which for most games, is the same thread on which the simulation logic is run. For an in-depth examination of this process, please refer to [2], which provides many significant and properly documented examples.

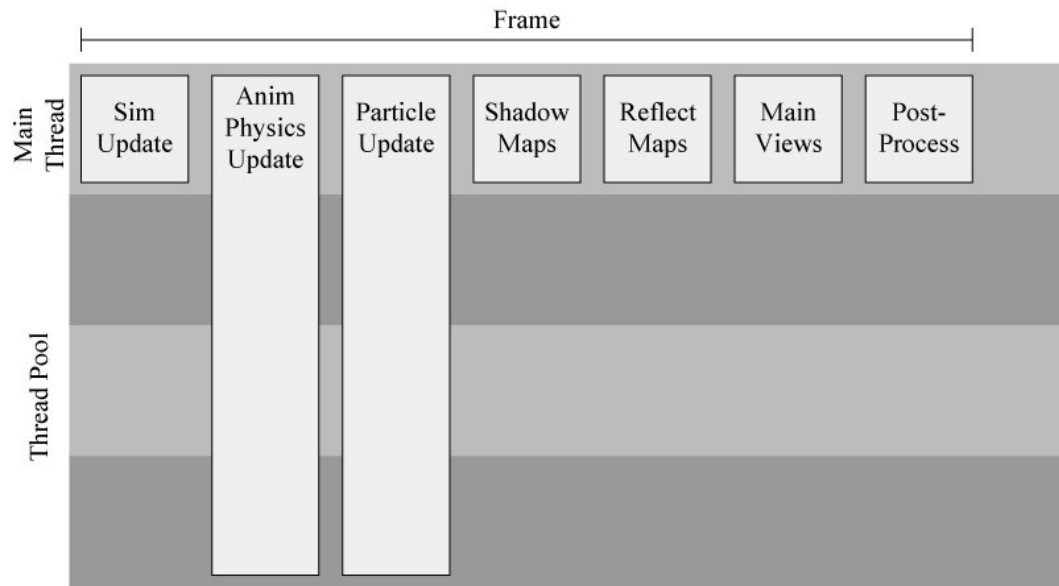


Figure 10.1: A standard game usage of parallel processing. Notice that only the update information tends to be multithreaded.

Figure 10.1 displays an example frame setup for a given game engine. In this example, the rendering device is owned by the primary thread, and as such, rendering a scene blocks the frequency of simulation updates. As described, the only two systems that use the thread pool are particles and animations, namely because of their ability to be updated without memory contention, and the concept of scaling back accuracy or LOD for these systems is trivial. The end of our frame goes through the process of rendering shadow maps, reflections, the main view, and finally post-processing.

Because of the fact that we must submit commands to the device on the thread that owns it, most of the rendering commands are interwoven with logic-based operations such as scene traversals, update logic, and the like. This type of rendering architecture can make it difficult to maintain your engine and port it to other platforms.

To be fair, modern engines are not architected as poorly as shown in Figure 10.1, and on average make much better use of thread pool availability. Some excellent alternative threading architectures for various genres of video games, most of which operate by offloading the render device ownership to a separate thread, are listed in [2]. This follows the observation that the simulation code is not required to update at the same frequency as the graphics rendering, and offloading the rendering to a different thread allows the simulation to continue on to process the next frame after submitting a draw request.

Although this architecture is more common now, it is still far from ideal. Even with the device submission being offloaded to a different thread, it would be beneficial to have the ability to modify the drawing part of a frame such that it could make better use of thread pool availability, thus balancing out your thread utilization more evenly.

10.2 Multithreaded Command Buffers

Modern rendering APIs operate internally with a data structure known as a *command buffer* containing rendering information needed to process draw commands at the device level. Typically, these command buffers are filled on your behalf by the exposed rendering API and inserted into a command queue that is executed at a later time. Because of this, the thread on which the rendering APIs are called incurs the overhead of filling the command buffer, often absorbing CPU cycles in a way that has historically been a common problem for modern games.

At the time of the writing of this gem, a few APIs allow the ability to create dynamic

command buffers or command lists on a thread separate from the one that owns the device, a process that allows us to distribute the overhead of filling the command buffers across the entire system. This represents an important step forward for rendering engines, as we now possess the ability to more finely control the workload of our rendering engine across our threading systems. The negative side, however, is that older APIs do not have the same abilities, causing issues for engines that aim to be compatible with multiple hardware levels.

To address this issue, Scheib [1] provided research showing a manner in which to create an overloaded DX9 device that would mimic the same recording ability as the newer APIs. To accomplish this, they overload the device APIs for the standard DX9 interface, and reroute them to their own system to composite a custom data structure that resembles an internal command buffer. They then submit this command buffer to the API on the primary thread owning the device using standard DX9 calls. On the primary thread, we are still incurring the overhead of submitting API calls, however due to the batched state, the overhead of API calls on the primary thread is reduced significantly, providing additional performance increases. Scheib shows that even with the overhead of submitting the buffer to traditional APIs, they still gain a significant performance increase by compositing the rendering command data on a separate thread.

10.3 Device-Independent Command Buffers

For legacy titles that would like to reduce the amount of API performance overhead without too much code rework, the method of [1] works quite well, allowing a drop-in solution that can benefit performance. For those titles with the luxury of analysis and not being bound by crunch deadlines, it's worth pointing out that the presented method can be considered overkill in terms of complexity, and short-sighted

in terms of API differentiation. Analysis of your rendering systems will prove that a subset of device APIs are often used, and more so that most of them occur in frequently predictable patterns. As such, when generating an API wrapper to the device, you waste time by offering support for functions that are not used by your title.

Logically, API calls should be hidden behind a wrapper interface, anyway, in order to minimize the amount of code that would need to be reworked to support multiple platforms and rendering APIs; as such, it makes sense that API calls for dynamic command buffer recording should also be hidden. In this context, the presented method of [1] is lacking, and a custom API-independent solution is required.

RenderCommand Structure

To this end, we present the concept of a `RenderCommand`, which contains the least amount of information required to submit a draw call to the API in a device-independent fashion. In effect, our `RenderCommand` structure is designed to mimic the draw commands that modern APIs use internally, containing information about which vertex buffer to use, the shading states, and how many polygons to draw. Depending on the needs of your graphics engine, the layout and members of this structure can vary greatly; however, in the interest of memory throughput, it's a good idea to keep this structure as small as possible by quantizing as much state data as you can. The code in Listing 10.1 shows an example of this process in that, rather than listing out explicit states for blending, a single enumerated state is used.

Listing 10.1: A simple `RenderCommand` structure that contains basic information for a draw call.

```
struct RenderCommand
{
    ResourceHandle  VertexBufferHandle;
    uint32         VertexDeclEnumIndex;
    uint32         NumTriangles;
```

```
uint32      NumVerts;

enum PrimType
{
    kTriList = 0,
    kTriStrip
};

PrimType    PrimitiveType;

enum BlendType
{
    kBlend_None = 0,
    kBlend_Standard,
    kBlend_Additive,
    kBlend_Subtractive
};

BlendType    BlendType;
// and so on...
}
```

The code in Listing 10.1 is a very stripped down, simplified version of what a full production `RenderCommand` structure would look like. For bonus points in the memory category, you could make a dynamically resizable version of this structure that holds only delta states that change between this draw call and the previous one. In effect, this matches closer to what the APIs use internally, but for the complexity involved, the simple version presented above will suffice for the descriptive purposes of this article.

Device-Independent Resource Handles

By design, the `RenderCommand` structure must contain handles to device resources that it will reference when executing the draw command. This exhibits a problem in

that directly exposing device handles to the rest of the systems makes it difficult to port the engine to other platforms, often causing rendering code to be spread out across the entire project. As such, it's often useful to create managers that hold the actual device-specific resource handles and offer a wrapped handle to the rest of the systems. These `ResourceHandle` objects can help the process of porting the code to other platforms and also grant you a buffer interface for doing asynchronous asset loading. A full discussion regarding the issues of multithreaded resource creation and management is beyond the scope of this gem. For more information, we refer the reader to [1] as an introduction to the topic.

Filling a `RenderCommand` Structure

A given `RenderCommand` structure simply references device information that it needs to execute. It does not, in contrast, actually load or own the device resources itself. As such, another class, which we will call a `RenderObject`, is responsible for managing the lifetime of a given resource (the actual resource itself should be owned by the manager responsible for it at a lower level, as described previously). Before filling in a `RenderCommand` structure, we assume that a given `RenderObject` structure has already been loaded into your engine and has communicated with the device in such a way to acquire proper rendering resources (such as vertex buffers).

Listing 10.2 describes the straightforward process of filling in a `RenderCommand` structure with the information contained in a `RenderObject` structure. As described, the `RenderObject` structure must contain information about how it needs to render, and in our simple example, it copies its state over to the `RenderCommand` structure based upon internal types and logic.

Listing 10.2: Filling a command buffer using generic handles. This is a great place to do additional logic related to rendering setup, since it will be executed on the thread pool.

```
void RenderObject::fillCommandBuffer(RenderCommand *RC)
{
    // make sure we're running on the threadpool
    ThreadAssert(ThreadPoolThread);

    if (ObjectType == kTypeOpaqueMesh)
    {
        RC->VertexBufferHandle = mVBHandle;
        RC->VertexDeclEnumIndex = kVD_Mesh;
        RC->PrimitiveType = kTriList;
        RC->BlendType = kBlend_None;
        RC->NumTriangles = numTrisFromPrimType();
        RC->NumVerts = mNumVerts;
    }
    else if (ObjectType == kTypeTransparentMesh)
    {
        // and so on...
    }
}
```

It's worth noting here that we're not computing new state for the `RenderObject` structure at this point; we're simply copying over state information relative to rendering and assigning it into the structure. This is a highly critical point when discussing the filling of these buffers on multiple threads, as this assignment pattern lends itself to being free of memory contention. That being said, the `fillCommandBuffer()` function is typically where your `RenderObject` structure would contain logic deciding *how* to fill in a `RenderCommand` structure properly. This includes things like checking the material to determine if we need to render with alpha blending. These types of logic and

data access patterns can get complex at times, which is why moving them off to a separate thread frees up cycles on your device thread.

Submitting a RenderCommand to the API

The act of submitting our custom `RenderCommand` structure to the API is overly verbose, yet direct in execution. Since we are filling in our commands on other threads, we must eventually resign ourselves to submitting the commands on the thread that owns the device. In practice, this relates to converting the data in our `RenderCommand` structure to information that we pass on to the rendering device APIs for execution. For more recent APIs that support command buffer creation, the act of submitting to the device requires a conversion from our custom command buffer to the desired device's format before submission; whereas for older APIs that do not directly support command buffer creation, the data must be directly fed to the device through API calls. Listing 10.3 describes the `executeDrawCommand*()` function, which at this point is the only function we've described that actually has direct access to the low-level rendering device. We present here the DirectX 9 version of the command, where we must call the API directly with our abstracted data types. It's worth noting that proper logic to determine which version of `executeDrawCommand*()` to call is a higher-level engine concept that is beyond the scope of this article; simple versions include a pointer to the proper function to call, while more advanced versions overload the `RenderControl` class entirely.

Listing 10.3: Submitting a `RenderCommand` structure to the API. This snippet of code is the only function that can actually communicate directly with the device.

```
void renderControl::executeDrawCommandDX9(const RenderCommand *params)
{
    ThreadAssert(DeviceOwningThread);
    // Set vertex stream
```

```
const VertexBufferContainer *vbc =
    mManagedVBs.getElement(params->vbHandle);
DX9Dev->SetStreamSource(0,
    (IDirect3DVertexBuffer9 *) vbc->devicehandle, 0,
    vbc->perVertSizeInBytes);

SetShaderData(params);
SetRenderStates(params);

// We use lookup tables for these mappings because it's faster.
DX9Dev->SetVertexDeclaration(StaticVDeclHandles[params->vDecl]);

D3DPRIMITIVETYPE type = PrimTypeMappingLUT[params->PrimitiveType];

// do draw
DX9Dev->DrawPrimitive(type, 0, params->NumTriangles);
}
```

Because our `RenderCommand` structure wraps up data and information in an abstracted fashion, submitting that data to the API has to include redirection from the `ResourceHandle` types to API handles that can be sent to the device. Listing 10.3 below shows this process directly, where the vertex buffer manager must be given a `ResourceHandle` object in order to receive the proper device handle.

10.4 A Camera-Centric Design

As we've discussed, a camera is the coarsest form of batching container used to gather work for rendering. So far, we've described a system that allows us to fill in single command buffers in a device-independent manner and submit them to the render device at a later time. Now, we need to describe the proper manner of creating container classes to aid with this process of batching `RenderCommand` objects as well as a larger

engine design to scale easily with multiple threads.

Balancing Rendering Across Multiple Threads: Everything's a Camera

With the ability to render across multiple threads, the next step is properly utilizing this feature and applying it to your threading system. This means you need to figure out how to create job packets that represent rendering work to be done by filling in the `RenderCommand` structures. At the most direct level, it makes sense to simply create one job packet for each draw call that would occur in your scene, thus creating one `RenderCommand` object per job. There are some issues with this, the foremost being that modern engines typically group many draws together into a single command buffer so that it minimizes API submission overhead and also takes advantage of things like redundant render state filtering. Creating the draw commands independently of each other robs us of the ability to take advantage of this optimization as the commands are being created.

As such, it makes sense that we still need to create draw commands in such a way to take advantage of state filtering by batching them sequentially. The difficult part about this process is determining how to properly batch up your `RenderCommand` objects to take advantage of this. Typically, most engine designs embrace command buffer generation for objects being rendered into the primary view, but not for subsequent things like rendering shadow maps. This results in an unbalanced throughput with your rendering pipeline, being that submission of some draw commands in one section are significantly faster than in others.

As an alternate view on the rendering process, a given frame of rendering in a game can be described as a grouping of cameras that all contribute their view data to the final scene. GPU-based shadow mapping is a great example of this concept in action, as it defines the same camera and render-target system that your primary view does, dealing with the same culling, redundant state overhead, and LOD. Reflective render targets,

offscreen particle buffers, and tiled antialiasing systems all exhibit the same characteristics as well and can be described using the same concepts.

This effectively solves a problem for us. Because our scene can be described in terms of cameras, we can use that idea to batch our `RenderCommand` generation, since objects that render to the same target typically share some sort of similar state data (for instance, shadow mapping requires all objects to use a different set of shaders).

Dividing our recording work by camera also provides us with a simple heuristic for scaling back workflow based upon hardware features. For instance, if we know the hardware is not fast enough to handle real-time reflections, cube-map cameras can be avoided and not processed. Another example is tiled antialiasing, a process in which subregions of the primary camera are used to generate larger images for portions of the screen, which are then downsampled and combined to generate the final image. By viewing each subregion as another camera render target, changes between antialiasing levels simply result in addition or removal of cameras from the system.

Figure 10.2 shows how a given rendering pipeline can change by breaking up command buffer generation by grouping them across cameras. Notice that the amount of work done across multiple threads increases (decreasing our overall processing time on the primary thread), but we add an additional API submit phase on the thread that owns the device.

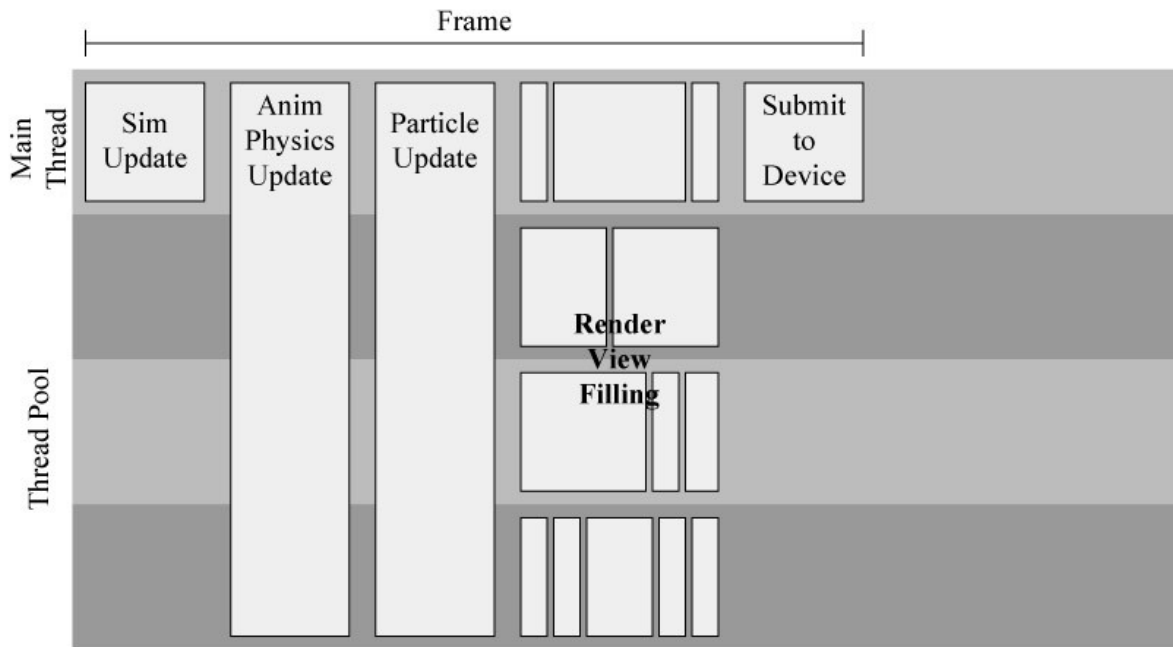


Figure 10.2: A camera-centric design. We make better usage of the thread pool for additional processing of rendering jobs.

For modern APIs, it's worth pointing out that the `RenderCommand` submit phase in Figure 10.2 is significantly smaller because the command buffers are simply copied from main memory to the device. This is possible due to the fact these APIs have their own command buffer formats, and a translation from our custom `RenderCommand` structure to the APIs version can be trivial. For older APIs that do not support command buffers, you cannot simply translate the command buffer data to the device format. Instead, you must communicate with the API through standard function calls as you normally would, using the data in the `RenderCommand` structure, as shown in Listing 10.3.

RenderView Structure

At this point, we seek to fill `RenderCommand` structures by logically dividing them into rendering bins based upon the cameras to which they are visible. This is beneficial to us, because it will allow us to group together these draw commands, which typically will share some sort of state data (at the very least, they will all share the same camera transform matrix).

A `RenderView` object defines a structure that contains a linkage between a camera and its render target. More importantly, a `RenderView` object describes how a given render target is filled, meaning it must contain a list of all objects that are to be rendered to that given render target.

Listing 10.4 shows the comparison between a `Camera` structure and our `RenderView` structure. Notice that in general, our render view is a superset of a camera, taking into account additional graphics-related properties. In addition, note that we still contain wrapped `ResourceHandle` objects to represent our destination render targets. The `RenderView` structure also contains a list of `RenderCommand` pointers, which are filled in by the `RenderObject` objects that are visible to this view.

Listing 10.4: Comparison between a `Camera` structure that the simulation would use, and a `RenderView` structure.

```
struct Camera
{
    Float3    at, up, right;
    float     aspectRatio;
};

struct RenderView
{
    Camera    ViewCamera;
```

```
Frustum          Frust;
RenderTargetHandle DestColorRTT;
RenderTargetHandle DestDepthRTT;

List<RenderCommand *> RenderCommands;

// this enumeration is very important as it defines the
// order in which we submit render views to the API
enum ViewType
{
    kVT_ShadowMap = 0,
    kVT_ReflectionMap,
    kVT_MainCamera,
    kVT_PostProcessing,
    kVT_Count
};

viewType          ViewType;
}
```

It's worth taking a moment to discuss the `ViewType` member of the `RenderView` structure. Although we're compositing view information on multiple threads, we still require a specific resource dependency as we're submitting primitives to the API. For instance, we need to composite all the shadow map data before the primary view so that objects that use those resources can rest assured that they are available for use. To accomplish this, we must tag each `RenderView` object with a type so that later on, we can submit the render views to the API in a proper, serial manner. We cover the implementation of this process in more detail below.

Filling a Render View Structure

Creating and filling the render views is a very simple process, but still requires a bit of understanding in the realm of threading, or at least an understanding of thread

pools. We refer back to a concept that a large portion of your environment can be represented as a camera view, and as such, we must iterate over those object containers to create our render views.

Listing 10.5 below covers the process of creating a new `RenderView` structure for each camera type in your scene. This includes primary cameras, shadow maps, reflection maps, etc. Once the views have been created, we create jobs for the thread pool that fill in a given render view. At this point, the order in which the `RenderView` objects are assembled is trivial since the data access patterns should already be thread safe.

Listing 10.5: Creating all the render views for a given frame.

```
void renderControl::CreateRenderViews()
{
    List<RenderWindow *>      currentViews;

    // for each primary camera (this includes portal cameras)
    for (int i = 0; i < mCameras.size(); i++)
    {
        currentViews.add(new RenderView(mCameras[i], kVT_MainCamera));
    }

    // for each shadow map!
    for (int i = 0; i < mLights.size(); i++)
    {
        if (mLights[i].IsShadowCasting())
        {
            currentViews.add(new RenderView(mLights[i].getShadowCamera(),
                kVT_ShadowMap));
        }
    }
}
```

```
// for each reflective target, etc...

// now fill our render views with visible objects in a
// threaded environment.
for (int i = 0; i < currentViews.size(); i++)
{
    Thread pool.QueueWork(procThreadedFillRenderView,
        currentViews[i]);
}

Thread pool.waitForWorkToFinish();
}
```

Once we've created our render views, we need to move forward with determining what objects to render. To do this, we must first cull the environment against the frustum of the camera owned by the render view, and then create a `RenderCommand` structure for each object that's visible to this camera. These `RenderCommand` objects can reside in a list structure that can be submitted sequentially to the API at a later time. Listing 10.6 covers this process by describing the internals of a thread procedure that creates the `RenderCommand` objects for a given render view. We assume that your object manager has some ability to perform frustum culling, from which you then gather the visible objects and have each one create a new `RenderCommand` structure.

Listing 10.6: Filling in `RenderCommand` structures should occur in a thread-safe manner, on a separate thread.

```
void renderControl::procThreadedFillRenderView(void *DataPacket)
{
    RenderView *currView = (RenderView *) DataPacket;
    List<RenderObject *> objects =
        gObjectManager.giveFrustumCollision(currView->frustum);
}
```

```
for (int q = 0; q < objects.size(); q++)  
{  
    RenderCommand *RC = new RenderCommand();  
    Objects[q]->fillCommandBuffer(RC);  
    currentViews[i].RenderCommands.add(RC);  
}  
}
```

It's once again important to point out that your data access model at this point needs to be a read-only system that is thread safe. This means that while traversing your object hierarchy to cull against a camera frustum, you should be doing so in a manner that does not corrupt memory for code accessing the same data in other threads.

For the sake of thread safety and memory coherence, we allocate a new `RenderCommand` structure for each instance of an object for each camera to which it is visible. Our particular example uses this allocation metric for a few reasons, of which one is the assumption that you will submit your `RenderCommand` objects in a thread separate from the one they are allocated in, requiring frame-coherent memory. This can be a very performance-heavy process, and as such, you should keep an eye on it in case you need to implement a custom container that has faster allocation/deallocation speed. If this does not match your particular threading system, then you may be able to get by with a less dynamic model of `RenderCommand` allocation.

Submitting a Render View to the API

Submitting a render view is a fairly straightforward process. We must simply bind the render target we're drawing to and then submit each of the commands that we have in our list to the API for drawing. Listing 10.7 shows this process, and it adds an additional set of data to indicate whether the render target should be cleared.

Listing 10.7: Serializing render views requires us to resolve them in a manner that satisfies dependencies.

```
void renderControl::serializeRenderViews(List<RenderView *> Views)
{
    for (int viewType = 0; viewType < Count; viewType++)
    {
        for (int i = 0; i < views.size(); i++)
        {
            if (Views[i].mViewType != viewType) continue;

            BindRenderTarget(Views[i]->renderTarget,
                Views[i]->DepthTarget);

            if (Views[i]->clearTargets)
            {
                ClearTarget(Views[i]->clearFlags,
                    Views[i]->clearColor, Views[i]->clearDepths);
            }

            for (int k = 0; k < Views[i]->commands.size(); k++)
                executeDrawCommand(Views[i]->commands[k]);
        }
    }
}
```

In addition, Listing 10.7 highlights a very important aspect of serialization of `RenderView` objects, namely that some `RenderView` objects are dependent on others, so even though we're compositing them in multiple threads, we must still submit them in a particular order to the API. For instance, you need to composite shadow maps before you can use them in subsequent draw commands.

10.5 Future Work

We've presented in this article a means in which to offload the overhead of creating command buffers to separate threads in a device API agnostic manner and use a camera-centric design to ensure proper load balancing across multiple threads. As we continue to take greater advantage of our rendering APIs for general purpose computing, the ability to properly break up work based upon packets of rendering data will continue to be important.

There are many project- and system-specific issues that are related to this process that are beyond the scope of this article. For completeness however, we briefly describe them here and leave the nitty-gritty details as an exercise for the reader.

Sorting and Instancing

`RenderCommand` objects provide excellent dynamic primitives for instancing evaluation. Once all the objects for a render view have been culled and added to a `RenderCommand` buffer, it is trivial at that point to define multiple types of sorting operations that can organize the `RenderCommand` objects in the buffer in various ways. Sorting by material index, vertex data, and object type often lends itself to a sorting process that combines objects in the command buffer in such a manner that multiple draw commands can be removed and a single command that uses instancing can be used. And if you're generating your `RenderCommand` structures dynamically, you can easily remove the original commands and insert the new instanced version. On some hardware, this may be a more performance friendly method of sorting by data, as your `RenderCommand` can take better usage of processor caches and reduce memory traversal that would normally be required when walking your object list.

Better Load Balancing

For those of you who are more proficient at threaded architectures, it's worth noting that there's a modification to this process that can take greater advantage of your threading architecture. The practical observation is that most camera-specific `RenderCommand` generation processes exhibit uneven processing times. For instance, smaller viewports or different types of API commands cause less work to be done. As such, you can often wind up wasting processing time waiting for these unbalanced threads to finish.

A more advanced solution is to modify your thread procedure to create one thread job for each draw call, allowing each to be created on a separate thread, yet filling in the camera data structure atomically. This usually means that your visibility culling for each camera would need to occur on the primary thread so that you can ensure proper allocation of space in your `RenderCommand` list. In order to take advantage of render-state filtering, sorting, or instancing, you must perform these processes once all the individual `RenderCommand` objects have been properly created.

The benefit of this modification is that you've now created smaller job packets that can maximize your thread utilization over the course of your frame. This is very useful on platforms where your thread operations can be interrupted by OS events. For a further discussion on issues related to small-job packets in video games, please refer to [3].

References

- [1] Vincent Scheib. "Practical Parallel Rendering with DirectX9 and 10". GameFest 2008.
- [2] Lindberg, et al. "Studies of threading success in popular PC games". Game Developers Conference, 2008.
- [3] Randall Turner. "Saints Row Scheduler". Game Developers Conference, 2007.

11

A GPU-Managed Memory Pool

Jeremy Moore

Black Rock Studio

Overview

The PlayStation 3 and Xbox 360 game consoles both contain unified memory architectures in which the GPU can directly read to and write from CPU-accessible memory. The graphics APIs on these consoles allow graphics data to be placed anywhere in memory and provide the ability to directly create and manipulate GPU resources such as textures or vertex buffers. With this low level of control, it is natural to consider the construction of streaming systems in which we dynamically load, move, and unload the resources that the GPU renders.

Streaming systems require that data is copied into memory where it can be accessed by the GPU for rendering. One implementation option is to manage all data copying with the CPU through a traditional `memcpy()` style API. Since the CPU and GPU run concurrently, we need to take care to ensure that any data is in place when the GPU is ready to read it. However, dealing with the synchronization overhead in achieving this can be complex and error prone.

In this gem, we describe an alternative solution that uses the GPU to manage the data copying. This approach ensures that the complexity and overhead of

synchronizing GPU operations and data movement are vastly reduced. In addition, this approach removes the CPU costs for data copying and, on our target platforms, allows us to achieve higher data copying bandwidths.

To illustrate this approach, we outline the design of the GPU-managed memory pool used for the streaming system implemented at Black Rock Studio. This system was written for use on our racing game *Split/Second*.

Note that the ideas outlined here are mainly relevant for our target console platforms that give a high degree of control over GPU resources. This isn't the case on the PC platform where the graphics API necessarily abstracts such details away from us.

11.1 Background

Streaming Requirements

The game worlds in *Split/Second* are large and richly detailed. We cannot fit all of the graphics resources used to render them within the fixed memory space of our target console platforms. The solution for this problem is to dynamically stream into memory only the resources needed to render the part of the world nearest to the camera.

Any streaming system consists of a number of components, such as the system to load resources efficiently from disc or the logic for deciding which resources should be loaded for a given camera position. In this gem, we concentrate only on the component that manages the memory used to store loaded resources.

Resource Types

There are a number of graphics resource types that we use when rendering. In decreasing order of typical size, some examples are textures, vertex buffers, index buffers, shaders, and constant buffers. On current consoles, these resources are each made up of two parts. First, there is a small fixed size component that is read by the

CPU, which we will call the "header". Second, there is a larger variable-sized component that is read by the GPU, which we will call the "data". When submitting rendering commands, the CPU parses the header to create entries in the GPU command buffer. The command buffer then contains references to the data that the GPU reads when executing the rendering commands.

In this gem, we only consider the management of the data that is read directly by the GPU. The management of the header is a simpler problem. It generally involves fixed sized objects for which a fixed sized object memory pool [2] is a common approach.

Design Requirements

When gathering the requirements for the streaming system in *Split/Second*, we decided that it should be able to dynamically load and unload any GPU resource from disc. We wanted to avoid overly complex code for synchronizing the GPU with the streamed resources. We also wanted to avoid the issues that can occur with memory fragmentation when resources are continually allocated and deallocated.

11.2 The Memory Pool

At the core of our streaming implementation is the memory pool class. We define our memory pool to be a block of contiguous memory along with the logic for dynamically managing it. We use one or more memory pool objects to manage the memory in which we store the GPU data resources for rendering.

A simple design choice would be to split the memory in our pool into a number of blocks with predetermined but variable sizes. Each new allocation in the memory pool would then use one of these blocks. This technique sidesteps fragmentation issues and reduces the memory management logic to simply finding an appropriately sized empty

block in which to place each asset.

This might be a good approach if we want to store only texture data, since textures tend to take one of a finite combination of sizes and surface types. However, it would waste memory when we cannot predict with good accuracy what range of block sizes we need to support. Our design requirements state that we want to support other resources in our memory pool such as vertex and index buffers. These have a large potential range of data sizes. We therefore choose to avoid any assumptions about resource size and do not use the fixed block approach to memory management.

Instead, our memory pool is made up of dynamically sized data "chunks". These chunks can be of any size, and may contain data or be empty "free chunks". Although a chunk may contain more than one item of data, each item of data is guaranteed to live within a single chunk. This means that chunks can be moved around the memory pool without breaking the internal consistency of the data that they hold.

Chunks need to be kept aligned according to platform restrictions. For example, on one target platform, all vertex buffer data may need to be aligned to 128-byte address boundaries. Often, there are different alignment restrictions for different resource types. For simplicity, all chunks can be aligned to the size of the platform's maximum data alignment restriction. For example if our target platform also requires that textures need to be 1 kB aligned, then we might chose to align all chunks, including those containing only vertex buffers, to 1 kB.

A memory pool starts out with no data in it and so contains a single empty chunk that spans the entire memory in the pool. When we add new data to the memory pool, we find an empty chunk that it fits inside. If the size of the new data is less than the size of the empty chunk then we break the chunk in two. One chunk now contains the new data, and the other is a free chunk containing any remaining unused memory. When we remove a chunk from the memory pool, we mark it as empty and merge the resulting

free chunk with any adjacent free chunks. A typical sequence of operations is illustrated in Figure 11.1.

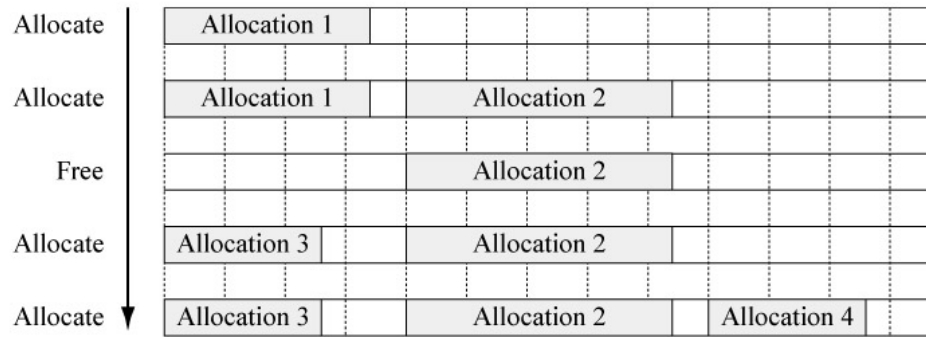


Figure 11.1: A memory pool layout shown as it undergoes a number of allocate and free operations. Note that each data chunk is kept to the alignment shown by the dotted lines. Also note that after only a few operations, we have a fragmented memory pool.

11.3 Synchronization Issues

On our target platforms, the GPU executes a single command buffer stream created by the CPU. The serial execution of these commands is the key to the memory pool data movement remaining synchronized with GPU usage.

Consider the case where we have a texture in the memory pool that we use for rendering, and then wish to move it within the memory pool before using it for rendering a second time. This is a typical sequence of operations if we wish to support defragmentation of the memory pool. If we use the CPU to manage memory copying, then the CPU is forced to wait for any rendering that uses the texture to complete before moving the texture. This is difficult to synchronize correctly and efficiently. It is also intrusive in that we need to keep track of when the texture is used. Figure 11.2 shows how this approach is executed on the CPU and GPU sides.

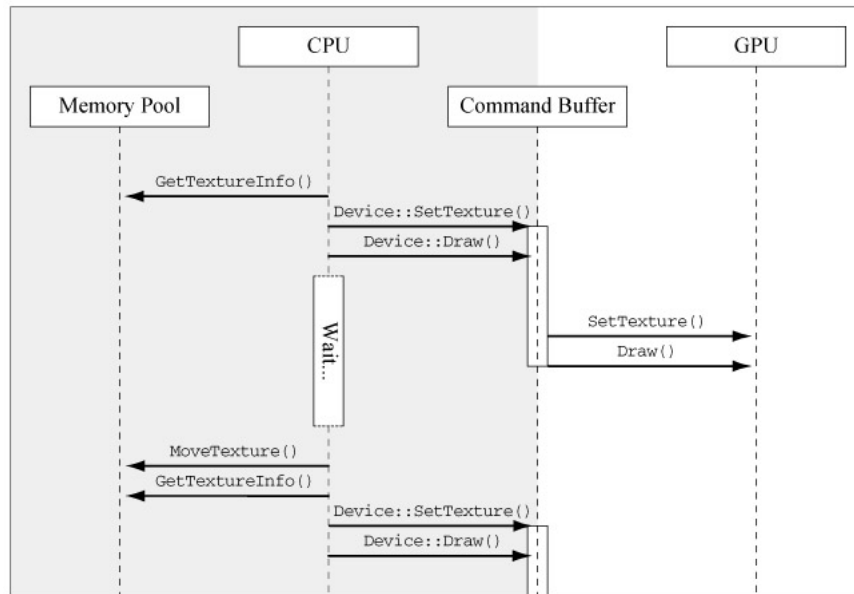


Figure 11.2: Sequence diagram of memory movement within our memory pool using the CPU. Note how the CPU needs to wait for an indeterminate length of time before moving the texture data in the memory pool.

In contrast, if we use the GPU to manage memory copying, then the GPU executes each movement of data in the memory pool as one action in a well-ordered stream of commands. This guarantees that any rendering from the texture is complete by the time we move the data. Conversely, it guarantees that the movement of texture data is complete by the time that it is next used for rendering. Figure 11.3 shows how this approach is executed on the CPU and GPU sides.

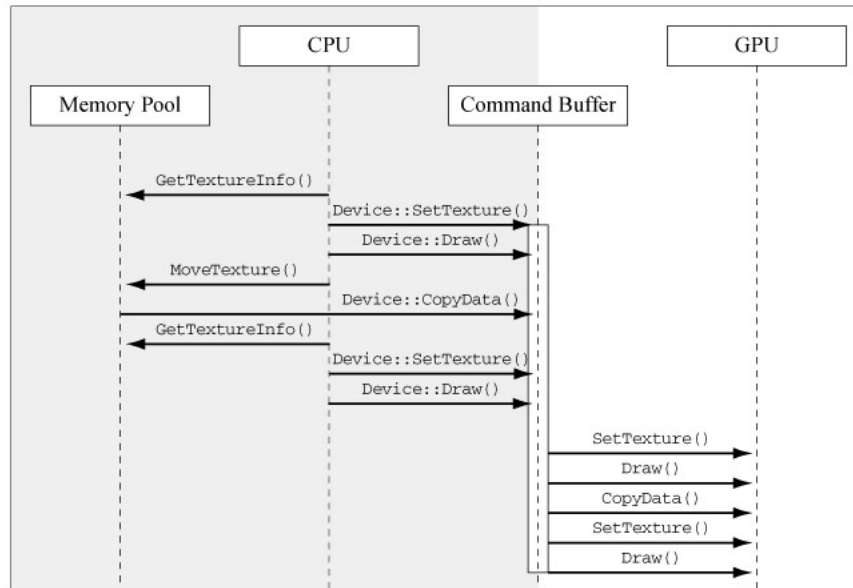


Figure 11.3: Sequence diagram of memory movement within our memory pool using the GPU. Note that no complex synchronization is now needed between the CPU and the GPU and that the CPU can queue the data transfer immediately.

11.4 The Staging Buffer

We have shown how the GPU can control memory movement within the memory pool. We also need to deal with the case of moving new data into the memory pool. When adding new data into the memory pool, we first search for a free chunk to place it in. A chunk labeled as free in the memory pool may be in one of two states. It may be genuinely empty or it may contain valid data that has been scheduled to be moved by the GPU. In the latter case, any data copying into the chunk by the CPU would potentially corrupt the data already in the memory pool. For this reason, only the GPU should copy new data into the memory pool. To accomplish this, we initially load any new data into a CPU-managed staging buffer and then use the GPU to copy this data

into the memory pool.

This approach still requires some straightforward synchronization between the CPU and GPU. In our implementation, the staging buffer is a ring buffer structure. The ring buffer contains blocks of data that are awaiting upload to the memory pool. We can only clear each ring buffer entry when the GPU has completed copying it to the memory pool. To track the progress of copies into the memory pool, we use a fence synchronization primitive to determine when the GPU has completed processing an operation. The fence primitive is named and implemented slightly differently on different platforms. On Xbox 360 and in OpenGL, it is called a *fence*, but in DirectX it is called an *event*, and on PlayStation 3 it is called a *label*. On each of these platforms, we can push a fence to the GPU command buffer and poll to see whether it has been processed by the GPU. So for each ring buffer entry in our staging buffer, we place a GPU fence after the GPU copy operation. Once per frame, we check the fences to determine which copies are complete and clear the ring buffer entries accordingly. This usage is illustrated in Figure 11.4.

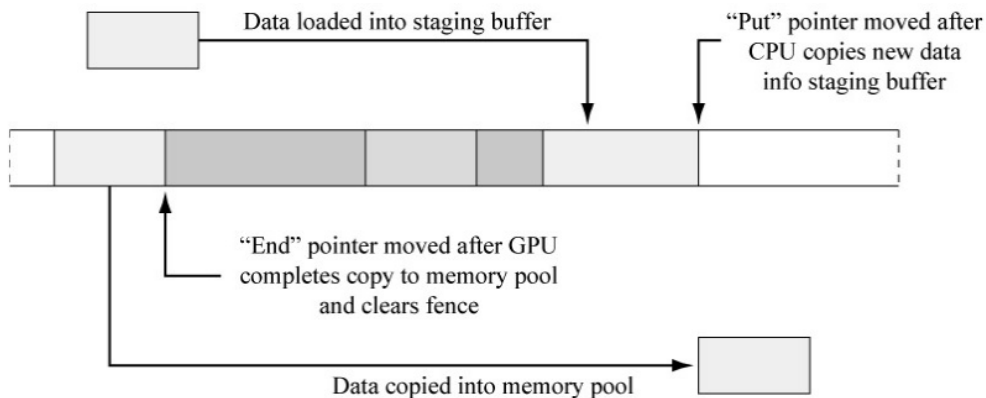


Figure 11.4: Staging buffer usage illustrating the use of fences to determine when a GPU copy is complete.

This staging buffer is part of the system that throttles data movement into the memory pool. In our implementation, we stream new data into the staging buffer using a background loading thread. If the data transfer into the memory pool stalls, either because of lack of space or slow defragmentation, then the staging buffer fills up and the background loading thread sleeps until space becomes available again.

11.5 Memory Pool Defragmentation

Since our memory pool contains many flexibly-sized chunks that are added to and removed from the pool in no fixed order, it is prone to fragmentation. Using the GPU to manage our memory pool frees us from concerns about multiprocessor synchronization. This makes any defragmentation system simpler to implement.

In their GDC presentation Balestra and Engstad [1] briefly describe a very simple algorithm that they employ to defragment streamed textures. Our initial defragmentation logic used the same approach. First, we scan through the memory pool from beginning to end until we find a free chunk. Then the next non-free chunk is moved down to fill this free chunk. The free chunk created by the move is then consolidated with any adjacent empty chunks and the scan is continued. This defragmentation pass is run once per frame. To avoid high GPU workload in pathological situations an upper limit is set on the total size of the data that each defragmentation pass can copy.

The advantage of this algorithm is that given a sufficient data transfer budget, we are guaranteed to tend towards a fully defragmented pool. A disadvantage is that defragmenting a single empty block at the start of an otherwise full memory pool requires an expensive copy of almost the entire memory pool.

In our naive implementation, we occasionally saw small spikes in the time it took

to load data into the memory pool. These generally corresponded to the poor defragmentation performance in the pathological case outlined above. A simple solution that reduced these issues to a manageable level was to break the memory pool into a number of regions. The defragmentation pass was then run for each region, but it never defragmented across region boundaries. The region size was tuned by testing to find the size that gave the best performance. This approach is illustrated in Figure 11.5.

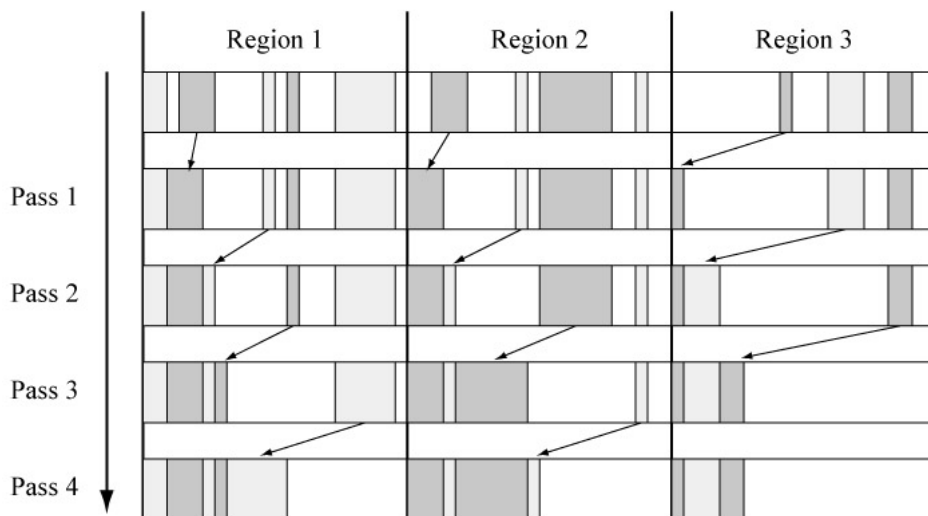


Figure 11.5: Memory pool layout shown over a number of defragmentation passes. The shaded areas represent allocated memory chunks. Splitting the memory pool into regions can reduce the number of memory copies required during defragmentation.

11.6 Memory Pool Eviction

When the memory pool is oversubscribed in our streaming system, we often need to select a candidate for eviction. Our memory pool logic implements this by assigning each resource in the streaming system an "eviction metric". The calculation of this

eviction metric is determined solely by application-specific logic. Once per frame, we evict the items in the memory pool with eviction metrics higher than the lowest eviction metrics of resources that are not in the memory pool. Because the GPU serializes all movement in the memory pool, eviction is simply a case of marking the evicted items' chunks as being free and therefore ready for reuse.

When calculating a sensible eviction metric, it is important to consider the possibility of cache thrashing. For an excellent summary of cache replacement algorithms suitable for a streaming memory pool see [3].

11.7 Platform-Specific Considerations

PlayStation 3

The PlayStation 3 memory architecture is split into main (CPU) and local (GPU) memory. The CPU has slow access to local memory, so it is important to place the staging buffer in main memory even if the memory pool is placed in local memory.

The PlayStation 3 has a very straightforward API for carrying out GPU memory copies. An important consideration is that some small care needs to be taken to ensure that a GPU memory copy of a resource is fully complete before that resource is used for rendering.

Xbox 360

The Xbox 360 has a unified memory architecture within which the GPU can be used to copy memory using its `memexport` API. However, when using `memexport`, the size of the memory pool that can be created and used is limited to 64 MB. Using the GPU to copy memory on the Xbox 360 has the advantage of having a significantly higher data throughput than when using the CPU. The GPU copies do bypass the CPU caches, however, so care needs to be taken to ensure that coherency is maintained with

any memory that the CPU directly accesses. For example, the staging buffer should either be in some form of non-cacheable memory, or when writing to the staging buffer, we should explicitly flush the cache lines down to memory.

11.8 Future Work

Multithreading Considerations

On our target platforms, an efficient method to balance the load of render call submission across multiple processing cores is to generate one or more command buffers per core that are combined and submitted to the GPU from a single main rendering thread. When doing this and using a GPU-managed memory pool, we need to take care that each core maintains a consistent view of the contents of the memory pool. This should be straightforward if the memory pool is updated on the main rendering thread at a time when no multithreaded submission is taking place, such as at the start or end of a frame.

Non-GPU Extensions

The pattern used in this gem might be extended to other processors that have the ability to move data in memory, but which operate concurrently with the CPU. The PlayStation 3 SPU is an example use case. This group of high performance processors can copy memory using a DMA and can be driven using an ordered queue of jobs. Many PlayStation 3 graphics engines use resources that are read only by the SPU to create input data for the GPU. The SPU could also be used to manage any memory pool containing these resources so that streaming and defragmentation logic is simplified.

Better Defragmentation

Although the defragmentation system outlined here is effective for our purpose, in the future, it would be worth researching the efficiency of other defragmentation

algorithms. A more efficient algorithm would have the advantage of reducing GPU memory bandwidth and removing potential stalls when loading data into the memory pool.

Acknowledgements

I'd like to acknowledge Balor Knight and Clément Dagneau, who both helped develop and integrate some of the ideas in this article. I'd also like to remember my colleague and friend Marek Romanowski who collaborated on some of these ideas, but who sadly passed away during the period in which I wrote this gem.

References

- [1] Christophe Balestra and Pål-Kristian Engstad. "The Technology of Uncharted: Drake's Fortune". Game Developers Conference, 2008.
- [2] Paul Glinker. "Fight Memory Fragmentation with Templated Freelists". *Game Programming Gems 4*, Charles River Media, 2004.
- [3] Colt McAnlis. "Efficient Cache Replacement Using the Age and Cost Metrics". *Game Programming Gems 7*, Charles River Media, 2008.

12

Precomputed 3D Velocity Field for Simulating Fluid Dynamics

Khalid Djado and Richard Egli

Centre Moivre, Université de Sherbrooke

This gem describes a method for simulating 3D fluid dynamics by using a precomputed velocity field. First, we present a method for building the fluid velocity field by using fluid dynamics. The fluid velocity field is computed on a fixed grid in the fluid domain. Second, we present a method for simplifying the fluid dynamics by using the precomputed velocities and some heuristics. The advantage to using a precomputed velocity field is that it reduces the fluid dynamic computation time. The greater part of the fluid dynamic computation is included in the velocity field computation process, which can be performed offline.

12.1 Introduction

In recent years, much effort has been devoted to integrating real physics into virtual worlds like those found in video games. Some of these techniques, such as collision detection, are now very familiar to game developers and have been widely integrated into game physics engines. This is not the case for fluids. Simulating fluid dynamics well in virtual environments is generally a challenge. The primary difficulty

is that there is no analytic solution to the Navier-Stokes equations describing the fluid dynamics. In general, the computer graphics community uses a grid or particle system to simulate a fluid. This gem uses an Eulerian 3D method on a grid to compute the fluid velocity field.

One of the first studies on simulating fluids in computer graphics was done by Foster and Metaxas [2]. Their work is based on a paper published by Harlow and Welch [3]. A great summary of work on fluids can be found in a recent book [1]. We have implemented the method of Foster and Metaxas [2] to compute the fluid dynamics, so the fluid domain is discretized into voxels. The velocity field computation process is as follows:

- The velocity source (or an exterior force), which we call the *blower*, is placed inside the fluid domain (for example, on a selected face of a voxel).
- The fluid velocity on all voxel faces in the fluid domain is calculated by solving the Navier-Stokes equations. These velocities can be stored in a file for future use, or precomputed before the simulation starts.

The use of fluid dynamics in video games is very expensive in terms of computing time. For greater performance, we precompute the physics of the fluid. The key idea is to precompute the steps that take a long time. Once the time-consuming steps in the calculation have been done, we use these data to simulate the fluid as if performing all calculations in real time. The method is thus fast, while yielding realistic and acceptable results for interactive applications such as video games. The opportunities afforded by the approach used in this article are:

- Computation of the fluid velocity anywhere in the fluid domain using the precomputed velocities.
- Simulation of a new velocity field using the precomputed velocities and heuristics when the blower changes intensity and direction.

- Simulation of the presence of a new object or the motion of an existing object in the fluid domain using the precomputed velocities and heuristics. This allows interaction between fluid and objects.

12.2 Velocity Field Computation

The velocity field represents the fluid velocity anywhere in the simulation domain. To obtain the velocity field, we start by computing the velocity on faces and then the velocity on voxels. Each voxel has six faces. The velocities on a voxel are shown by the red lines and the velocities of faces by the green lines in Figure 12.1. We use the finite difference method described in [2] to compute the fluid velocities on faces and voxels. The blue line in Figure 12.1 represents the blower velocity introduced in the domain. The Navier-Stokes equations are

$$\begin{aligned} \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} + w \frac{\partial u}{\partial z} &= \frac{\partial P}{\partial x} + g_x + \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \\ (12.1) \quad \frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} + w \frac{\partial v}{\partial z} &= \frac{\partial P}{\partial y} + g_y + \nu \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2} \right) \end{aligned}$$

$$\begin{aligned} \frac{\partial w}{\partial t} + u \frac{\partial w}{\partial x} + v \frac{\partial w}{\partial y} + w \frac{\partial w}{\partial z} &= \frac{\partial P}{\partial z} + g_z + \nu \left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2} \right) \\ (12.2) \quad \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} &= 0 \end{aligned}$$

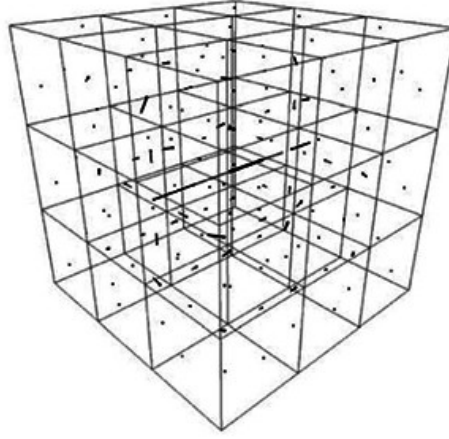


Figure 12.1: (See also Color Plates.) Voxel and face velocities.

Using Equation (12.1), Equation (12.2), and the product rule, we obtain

$$\begin{aligned}
 (12.3) \quad & \frac{\partial u}{\partial t} + \frac{\partial}{\partial x}(u^2) + \frac{\partial}{\partial y}(uv) + \frac{\partial}{\partial z}(uw) = -\frac{\partial P}{\partial x} + g_x + v \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \\
 & \frac{\partial v}{\partial t} + \frac{\partial}{\partial x}(vu) + \frac{\partial}{\partial y}(v^2) + \frac{\partial}{\partial z}(vw) = -\frac{\partial P}{\partial y} + g_y + v \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2} \right) \\
 & \frac{\partial}{\partial t} + \frac{\partial}{\partial x}(wu) + \frac{\partial}{\partial y}(wv) + \frac{\partial}{\partial z}(w^2) \\
 & \quad = -\frac{\partial P}{\partial z} + g_z + v \left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2} \right)
 \end{aligned}$$

These equations were used in Foster [2] and for implementation in this paper. The quantities appearing in the equations are summarized as follows:

- u , v , and w are the velocities of the fluid on a face or a voxel in the x -, y -, and z -axis directions, respectively.

- P is the internal pressure of the fluid (formally, P is the pressure divided by the density of the fluid [4]).
- ν is the kinematic viscosity.
- \mathbf{g} is the acceleration of gravity.

Solving Equation (12.3) by the finite difference method gives the new velocity on each face of the fluid domain, like the velocities in green shown in Figure 12.1. These solutions can be found in [2] and in the source code provided with this gem. Note that the components of the velocity for faces perpendicular to the three axes (x , y , and z) are computed separately. Equation (12.2) is used to compute the null divergence of the fluid velocity. This process leads to fluid velocity correction and, finally, the pressure update.

To ensure the stability of the solver, the time step Δt and velocities need to satisfy the condition

$$(12.4) \quad \max \left\{ u \frac{\Delta t}{\Delta x}, v \frac{\Delta t}{\Delta y}, w \frac{\Delta t}{\Delta z} \right\} < 1$$

The steps taken by the solver are summarized in Listing 12.1.

Listing 12.1: Pseudocode of the full solver.

```

1 Simulate(const float& deltaT)
2 {
3     // Reset Velocities (Boundary and Blower)
4     ResetBoundaryAndBlower();
5     // Compute New Face Velocities on each voxel with Equation (12.3)
6     for (int voxelID = 0; voxelID < m_voxelNumber; voxelID++)
7     {
8         UofFaceVelocity(voxelID, deltaT);
9         VofFaceVelocity(voxelID, deltaT);

```

```
10     WofFaceVelocity(voxelID, deltaT);
11 }
12 // Null Divergence step using Equation (12.2)
13 float maxDivergence = s_epsilon;
14 while (maxDivergence >= s_epsilon)
15 {
16     float currentMaxDivergence = -INFINITE;
17     for (int voxelID = 0; voxelID < m_voxelNumber; voxelID++)
18     {
19         float divergence = 0.0F;
20         ComputeDeltaPressure(voxelID, deltaT, &divergence);
21         if (divergence >= currentMaxDivergence)
22         {
23             currentMaxDivergence = divergence;
24         }
25     }
26     maxDivergence = currentMaxDivergence;
27     // Pressure Update
28     UpdatePressure();
29 }
30 // Compute Voxel velocities by interpolation
31 ComputeVoxelVelocities();
32 }
```

To simplify the fluid dynamics, we precompute the velocities of each face for a single blower in different directions such as the positive x direction (see Figure 12.1). These precomputed velocities can be stored in memory before using the simplified simulation with heuristics. They could also be saved on disk and then loaded into memory when needed.

12.3 Physics Simplification

Using the full fluid dynamics solver as presented in Section 12.2, we can change blower direction and move an object. In this section, we present heuristics that allow us to simulate the presence and motion of an object and also to change the blower velocity direction without using the full solver. We have two kinds of heuristics: the "obstacle heuristic" and the "blower heuristic".

Obstacle Heuristic

With full calculation, an obstacle such as the black box shown in Figure 12.2 requires us to recalculate all quantities (velocities and pressure) in the fluid domain at each time step. The presence of an obstacle in the fluid domain entails a distortion in the velocity field. We simplify the full solver by not computing lines 4 to 11 in Listing 12.1 and by setting the number of iterations for the null divergence step. In fact, the divergence minimization process from lines 16 to 29 is in some cases performed more than 10 times when we set $s_epsilon$ to 0.0001. In the case of the "obstacle heuristic" we set the number of iterations to around 2 to make the process faster.

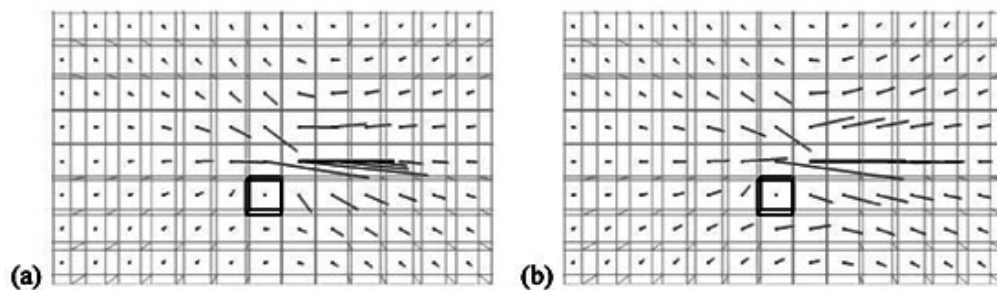


Figure 12.2: Images of the velocity field—(a) from the full solver; (b) from the obstacle heuristic.

The new velocity is computed by the interpolation described below. Let \mathbf{V} be the current velocity on a face, let \mathbf{V}_p be the precomputed velocity on the same face, and let c be a constant that is set to determine how fast the velocity field is returned to the precomputed state when an obstacle is removed. Δt is the time step. The interpolation is calculated using

$$(12.5) \quad \mathbf{V} \leftarrow (1 - c\Delta t)\mathbf{V} + (c\Delta t)\mathbf{V}_p$$

The computation of a null divergence field allows velocity modification around the obstacle. Null divergence means that the fluid flowing in equals the fluid flowing out. In fact, the velocities on the faces of the obstacle are zero, so the null divergence ensures that the fluid on the neighboring voxels gets around the obstacle.

The "obstacle heuristic" steps are summarized in Listing 12.2.

Listing 12.2: Pseudocode of the obstacle heuristic.

```

1 SimulateObstacleHeuristic(const float& deltaT)
2 {
3     // Update Velocities using the Precomputation
4     // for each face of each voxel
5     FaceVelocity = (1.0F - cstObstacle * deltaT) * FaceVelocity
6         + (cstObstacle * deltaT) * PreCompFaceVelocity;
7
8     // Null divergence step in Precomputed Version
9     PreCompDivergence(deltaT);
10
11     // Compute Voxel velocities by averaging
12     ComputeVoxelVelocities();
13 }
```

The main advantage of using the "obstacle heuristic" is that we are able to add and

move objects in the fluid domain, starting from the precomputed velocity field without any obstacle. The results obtained with the heuristic (see Figure 12.2(b)) are similar to those of the full Navier-Stokes solver (see Figure 12.2(a)), the advantage being that the process is at least twice as fast.

Blower Heuristic

In the fluid simulation, any change in the blower velocity direction means all quantities (velocities and pressure) need to be recalculated in the fluid domain at each time step. We simplify the full solver to be able to simulate a dynamic blower.

By observing fluid simulation and changes in velocity, we notice that the velocity is linear with the norm of the blower velocity. This means that if we double the velocity for the blower, the resultant velocities on the other voxels are also doubled. We also notice that when the blower changes direction, each velocity of a voxel changes direction. To be able to simulate the same effect, we use precomputed velocities by aiming the blower in the directions of the positive and negative coordinate axes. In this article, we use blower directions only in two dimensions along the x - and y -axes, but the method can be generalized to three dimensions. We have to precompute the velocity field for the blower in directions $(1,0,0)$, $(-1,0,0)$, $(0,1,0)$, $(0,-1,0)$. Let θ be the angle between the new blower direction and the vector $(1,0,0)$. For θ between θ_1 and θ_2 , we proceed as follows:

- We identify θ_1 and θ_2 according to θ . For example, $\theta_1 = 0$ and $\theta_2 = 90$ if θ is between 0 and 90 degrees.
- We compute the interpolation weight $f = (\theta - \theta_1) / (\theta_2 - \theta_1)$.
- \mathbf{V}_{p1} is the precomputed face velocities for θ_1 .
- \mathbf{V}_{p2} is the precomputed face velocities for θ_2 .
- Each face velocity \mathbf{V} is computed by $\mathbf{V} = (1 - f)\mathbf{V}_{p1} + f\mathbf{V}_{p2}$.

This heuristic is an approximation of the velocities yielded by the full solver version. Figure 12.3 shows a comparison of velocities from the "blower heuristic" and the full solver. The simulation using the heuristic is more than twice as fast as the full solver.

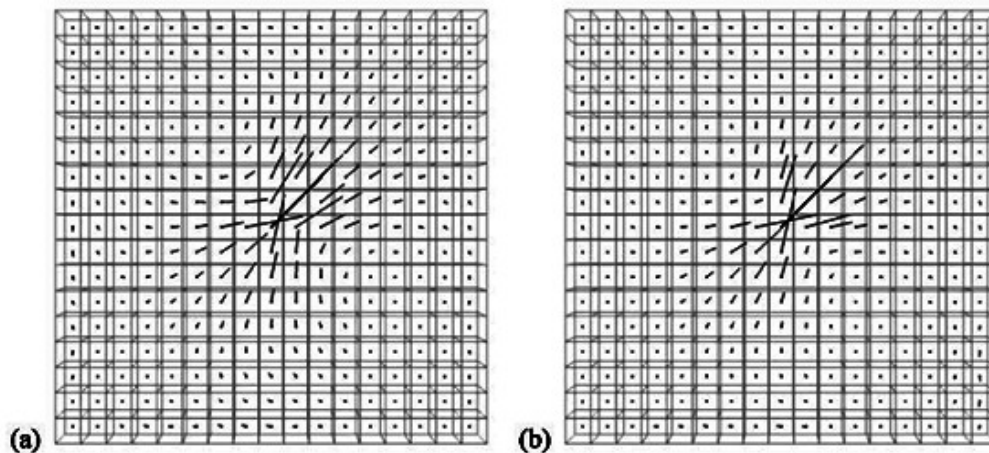


Figure 12.3: Images of the velocity field—(a) from the full solver; (b) from the blower heuristic.

12.4 Results and Discussion

To visualize the fluid velocity field, the velocities on voxels can be displayed as vectors. We are also able to visualize the velocity field using unit vectors for the direction and colors for the magnitude. For example, the velocity can be depicted in blue for a high magnitude or red for a low magnitude. It is also possible to visualize the velocity field with particles moving in the fluid domain. The Figure 12.4 shows examples of visualization methods.

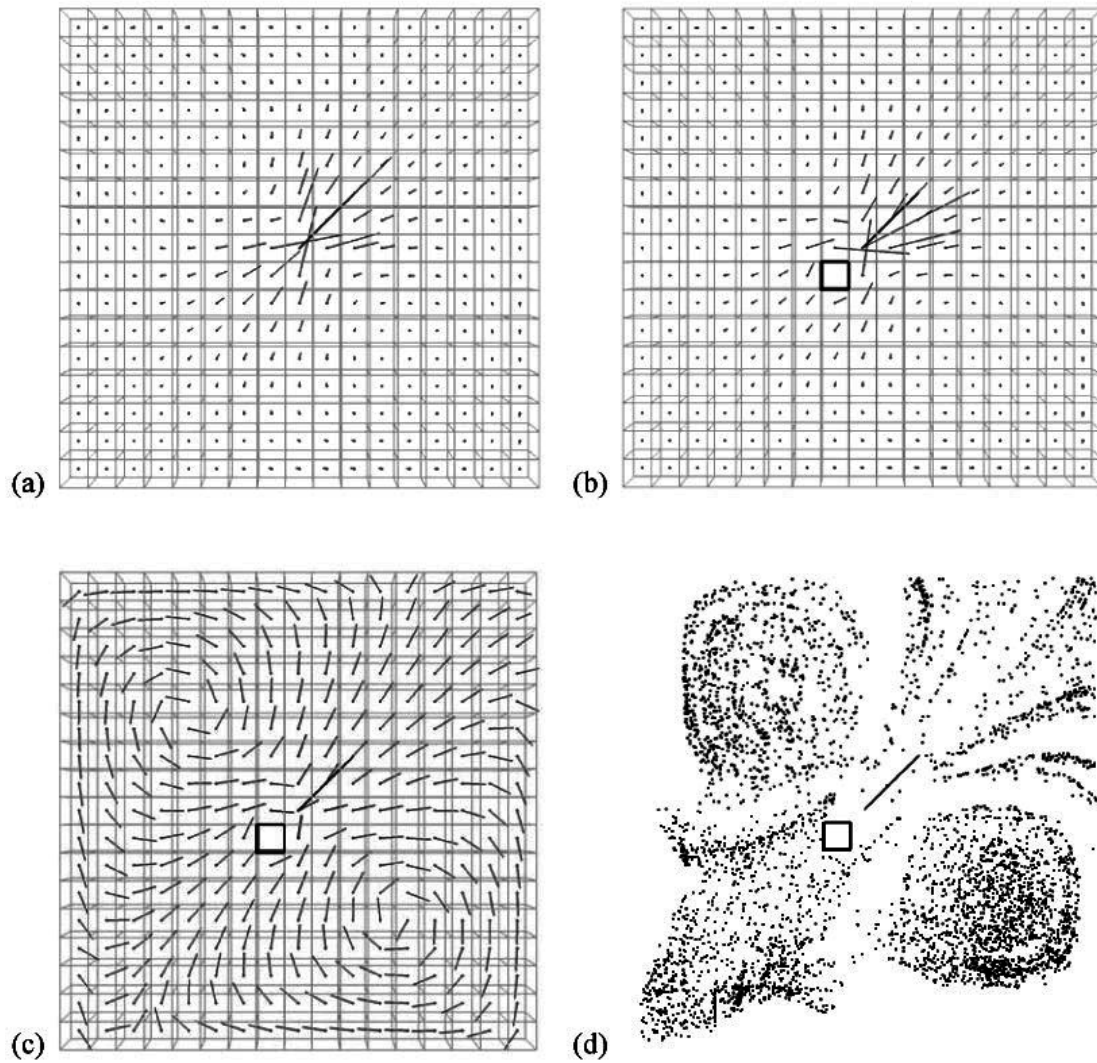


Figure 12.4: (See also Color Plates.) Images of the velocity field visualization using heuristics—(a) with vectors without an obstacle; (b) with vectors with an obstacle; (c) with unity vectors for the direction and color for the amplitude with an obstacle; (d) with an obstacle and particles.

To illustrate how the heuristics work, an implementation is provided on the accompanying CD. The program is written in C++ and uses OpenGL to display the 3D scene. The user must set the resolution of the fluid domain in terms of number of voxels. In the case of Figure 12.4, the fluid domain has $17 \times 17 \times 17$ voxels and the blower (in blue) is at the voxel position (8, 8, 8). The grid in Figure 12.4 represents only the voxels with $z = 8$ in the fluid domain since it is not easy to visualize the velocities with vectors when all $17 \times 17 \times 17$ voxels are displayed in a still image.

The full solver frame rate is around 172 FPS on a laptop equipped with an Intel CPU T2400 at 1.83 GHz (dual core), with no parallelism in the simulation and visualization processes. The same scene using "blower heuristic" (see Figure 12.4(a)) allows a frame rate around 392 FPS. We get 331 FPS with the two heuristics (see Figure 12.4(b)).

Some other optimizations are possible in a game physics context. For example, we don't have to update the velocity field when the blower doesn't change and the obstacle doesn't move for a certain time. The heuristics of this article can be simply added to an existing game physics engine. The velocities can be precomputed at setup or loaded from a file.

References

- [1] R. Bridson. *Fluid Simulation for Computer Graphics*. AK Peters, 2008.
- [2] N. Foster and D. Metaxas. "Realistic animation of liquids". *Graphical Models and Image Processing*, Volume 58, Number 5 (September 1996), pp. 471–483.
- [3] F. H. Harlow and J. E. Welch. "Numerical calculation of time-dependent viscous incompressible flow". *Physics of Fluids*, Volume 8, Number 12 (1965), pp. 2182–2189.
- [4] L. Quartapelle. *Numerical Solution of the Incompressible Navier-Stokes Equations*. Springer, 1993.

13

Mesh Partitioning for Fun and Profit

Jason Hughes

Steel Penny Games, Inc.

Overview

There are many situations in which an entire mesh is too much data to process—whether it's a CPU, SPU, or GPU, there are performance limitations to consider. In general, artists can do this work by hand, but human variability being what it is, a good algorithm is faster and more reliable, and it improves artist productivity. A key trait of a solid tools pipeline is its ability to free the artists from such burdens anyway. There is a situation in almost any 3D game in which one mesh really would work better as many smaller chunks that can be uniquely identified by the CPU and processed independently from others.

Specifically, what kind of limitations do real-world games run up against?

- Rendering limitations, such as 8-bit or 16-bit indices limiting the number of vertices that can be put into a mesh.
- Skinning limitations, such as the number of matrices a particular graphics chip is able to express with shader constants, or perhaps older fixed function pipelines that have a hard limit for the number of matrix indices per render call.
- Vertex unit bottlenecks, where a mesh has a few visible triangles but the majority are being transformed and rejected by the clipping or backfacing unit. The

bandwidth for transferring vertex data to the GPU is significant on certain architectures, and the vertex shader is potentially a bottleneck as well.

- Virtually any other operation that has a per-triangle component could potentially be improved by partitioning a mesh, especially if there are trivial rejections that could be performed on those partitioned mesh fragments as a whole.

13.1 Desirable Algorithm Properties

Now that we have some understanding of when splitting meshes into reasonable-sized chunks if helpful, is any partitioning mechanism good? If not, what are the ideal properties of a battle-hardened mesh partitioner? The following is an unorganized list of properties that I have determined through experimentation. Other properties may exist for certain kinds of games. Certain differences in hardware may shift the importance of some properties. Use your best judgement. Once you have a set of properties, you can define an objective fitness metric for how well the algorithm is performing. A fitness metric is crucial during the experimentation phase of algorithm design because otherwise, determining whether changes are beneficial, detrimental, or irrelevant is very time consuming and subjective.

Partitions Should Have Relatively Same-Sized Bounding Columes

Rationale: This improves culling performance since regularly sized partitions have a more predictable overhead per partition. It also means a rasterized set of triangles is likely to have a more consistent pixel throughput per draw call, allowing you to balance per-partition work versus per-triangle work by adjusting the maximum bounding volume.

Partitions Should Have Relatively the Same Number of Triangles and Vertices

Rationale: This improves predictability of bandwidth and transfer times to

dedicated processing units like the SPU by simplifying buffer management. It also smoothes out performance spikes that may otherwise occur. It's good for leveling out DMA transfer performance and improving the culled-to-rendered vertex ratio, which reduces total bandwidth to the GPU.

The Number of Partitions Should Be Minimized Overall

Rationale: Management costs per partition are often high, so reducing the number of partitions can only improve CPU performance. However, if this property is overemphasized, you end up with a single partition containing all the geometry—no partitioning at all.

The Number of Triangles Per Vertex in Each Partition Should Be Maximized

Rationale: A triangle requires three vertices. If you insert an adjacent triangle (sharing an edge) in the same partition, the new triangle only adds one vertex. However, the same adjacent triangle placed in a different partition creates three vertices. Obviously, you want to ensure as much sharing of vertex data as possible. Some duplication of vertex data is unavoidable because partitions naturally separate adjacent triangles along their borders. Another way to describe this is minimizing the number of borders between partitions, but that is more complicated to measure.

The Partitioner Should Guarantee a Solution With the Previously Described Properties in Predictably Bounded Time

Rationale: As a practical matter, it is unacceptable for a partitioning to take more than a few seconds because it hurts artists' ability to iterate. This demands a few data structures, some diligence about avoiding any possibility of infinite loops, and some thought to worst case scenarios that can "never happen".

13.2 Lessons Learned

My experience with building mesh partitioners over several years led me down many dead ends. Here are the biggest mistakes that I made, in no particular order, and what made them poor choices.

- Mesh data is not to be trusted from any source. The likelihood of any assumption regarding triangular mesh conditioning will be proven erroneous asymptotically approaches certainty near important milestones. You can never have enough asserts in your code to help diagnose these issues. In the end, my partitioner generated a completely different topological representation of the triangle data to be used solely for partitioning.
- Adding a triangle to an existing partition may add between zero and three new vertices. Partitions hold references to triangles, which imply vertices. Do not try to build partitions out of vertices, and construct the triangle set from them. You end up with duplicated triangles, heavily disproportionate partitions, and all manner of other issues. Since the important measurement of a partition is how many vertices are inside it (and whether a new triangle adds any new vertices), you must compute final vertex sharing before entering the partitioner.
- Do not build a partitioner that ever subdivides *and* merges the same partition in alternation. Do one or the other, or do them in sequence, but do not alternate between them. There are unforeseeable infinite loops, no matter how you craft the logic to prevent it from happening.
- Merging partitions seems like a good idea, at first. Initializing one partition per triangle and merging nearest neighbors was one method I discarded quickly. It is essentially Kruskal's minimum spanning tree algorithm [1]. However, Kruskal's approach cannot deal with user-defined limits for partitions, and breaks down quickly once you have an entire population of partitions that are 51% of your threshold. At this point, merging any two together means you have tiny leftovers that need to be put somewhere else—this means merging and partitioning in alternation,

causing infinite loops. Alternatively, you can leave those abandoned triangles for a final partition, which invariably violates every important property described above.

- Assure that the *fitness metric*, a function responsible for measuring the current "fullness" of a partition, is monotonically increasing as triangles are added. To explain further, one algorithm I attempted worked by stealing triangles from neighboring partitions when the neighbor was larger. The function that selects which partition to steal from neglected to test the fitness of source *and* target partitions after transferring a triangle. Certain parts of the data was compressed, and the size of the compressed data depended on the range of values present in the data itself. Most of the time, removing data from a partition caused its compiled packet size to shrink, but occasionally, it would grow. As a result, a situation would occur where *A* steals from *B*, then *B* steals from *A*, because whichever direction the triangle moved, the sizes of final data packets would flip-flop.
- Design for clear termination conditions and steady performance. I tried partitioning in which I randomly assigned triangles to partitions, then "shoved" poorly connected triangles to adjacent partitions that would mutually benefit both partitions, either in bounding volume reductions, packet size, or other metrics. It eventually degenerated to a pseudo-linked list traversal per triangle, where one triangle is pushed to a neighbor who is now violating some metrical limit and now must push a different triangle to another adjacent partition, and so on. Even when marking a path behind you to prevent loops, this tends to create huge linked list traversals through all partitions and slows down dramatically as the partitions begin to converge.
- Use appropriate data structures, and find ways to cache or reduce lookups for relationships that are costly to determine. My favorite is the simple adjacency list representation for graphs. It is trivial to implement and very easy to use when debugging algorithms.
- Ask the right question. For partitioning, that question is "Which triangle should I put in the current partition now?". Many of my attempts were trying to decide "Which triangle would be better to move from partition *A* to partition *B*?", "Which partition should grow?", or "Which partition should shrink?". Involving relative decisions

about the quality of partitions never materialized into a concrete and usable system for partitioning.

- Always select the least-connected triangle as the starting point for new partitions. This discourages your final partition from including a large number of scattered "loner" triangles that no other partition wanted. The bounding sphere of such a partition would be very large and the rendering very inefficient. This same rule applies for triangle stripping algorithms, for the same reasons.
- Graph theory helps. Read up on Prim's algorithm [2]. It is a template for how this mesh partitioner works. However, do not be tempted to follow Prim's algorithm exactly as described. It suggests a priority queue for candidate edges, but does not allow for reprioritizing adjacent nodes in the queue. Mesh partitions have potentially shared data between triangles, so it is likely that each additional triangle added to a partition changes the cost calculation for every candidate triangle. Placing triangles into a queue with a fixed priority clearly does not work for this kind of problem. This means you must scan the candidate triangle list every iteration and recompute the cost to find the best candidate.
- It is very hard to come up with a local metric for packing faces into tight clusters. If you put no restrictions on the closeness of triangles, you get long strips with large bounding spheres that poorly approximate the partition. If you measure relative to a centroid of faces, you get partitions that are very densely packed, but may not share vertices well (imagine three parallel planes intersecting a sphere, where triangles inside the sphere are close together, but have a lot of boundary vertices that are only used once). Just *after* I'd solved this problem, I read a paper [3] that expressed the same solution only a couple of years earlier. As with most discoveries, in retrospect, the solution is obvious: minimize the distance between related triangle centers.

13.3 When Greedy Is Good

Since there are so many choices that can be made while writing a mesh partitioner, it's surprising how simple a good one can be. Although the following is a greedy algorithm, the results are initially good, and a simple refinement step afterwards can

improve the partition efficiency by two to three percent. This section describes what I found to work well.

Core Algorithm Overview

1. Select the least connected ^[1] unassigned triangle and initialize a new partition with it. If no triangle is unassigned, partitioning is complete.
2. Collect all the unassigned triangles that are related ^[2] to the partition into a candidate list. If no related triangles exist, perform an exhaustive search to find all minimally connected triangles in the unassigned mesh, and consider them all to be candidates.
3. Iterate over the candidate list of triangles, temporarily adding each one to the partition, and compute the *fitness metric* of the partition using an objective metric function. Immediately reject any candidate triangle that causes a hard threshold limit to be exceeded, e.g., vertex count limit, final packet size ^[3], maximum bounding sphere.
4. If at least one candidate triangle was not rejected, select the triangle that yielded the best fitness score for the partition and add it to the partition. Repeat from Step 2.
5. Otherwise, there are no triangles that can be added to the partition without exceeding some threshold (bytes, number of triangles, bounding sphere, etc.). Consider this partition full and start a new one. Repeat from Step 1.

Refinement

After all triangles have been assigned to partitions, the last partition is, on average, half the size of the others. Even if this is acceptable, you might want to perform a refinement on the partitions. You can typically run the refinement multiple times to level out some of the partitions and reduce the total size of the solution. To refine, follow these steps for each partition:

1. Determine if any triangle can be moved from partition A to partition B, where A loses some vertices and B gains none. This is a clear win for memory and performance, but may distort bounding volumes or cause the triangle counts across

partitions to become unbalanced. This is demonstrated in step 14 of the graphical walkthrough at the end of this gem.

2. Determine if any triangle can move from partition A to partition B, where B has fewer triangles, even if the number of vertices remains the same in both partitions. This balances the triangle count between partitions, but can be done to improve the bounding volume or triangle count balance.
3. As a last resort, determine if any one vertex can be moved from partition A to partition B, possibly moving several triangles to B in the process, so that A has fewer vertices and triangles, and B has more. This is particularly useful when filling out the final partition, because the final partition is on average half the size of the others.

^[1]Least connected, in this context, means the unassigned triangle that has fewest shared vertices with other unassigned triangles. This is a dynamic property of a triangle during partitioning, and cannot be precomputed.

^[2]Related, as defined for partitioning, is the statement that vertex data within a partition could be shared with adjacent unassigned triangles. This property forms the basis of candidate triangles for inclusion in a growing partition.

^[3]In most modern graphics engines, a single batch of geometry sent to the GPU is called a *packet*. The size of this packet is often hard-limited at a specific number of bytes dictated by hardware or engine software design constraints.

13.4 Future Work

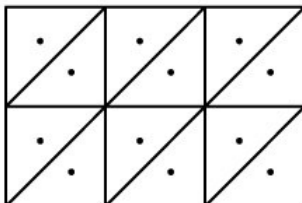
There are many interesting applications that a partitioner can be used for, once you have it. What kinds of rapid backface culling could you do on a large mesh if you partitioned it into triangles that all have relatively the same face normal with minor deviations? The speedup from bulk triangle rejection could be dramatic, without having to even transmit the triangles to the GPU.

Similarly, there is value in extending the partitioner to pay attention to the size of packets generated so that they fit inside a fixed memory size. This fixed size might make data management simpler and faster because less bookkeeping is necessary, particularly for an architecture like the PlayStation 3 where SPU memory is tight and double-buffering DMA is mandatory for best performance. Dynamically managing memory is slower and more complicated than using fixed buffers, so leaving room in your implementation to extend the fitness function of a partition in arbitrary ways is valuable.

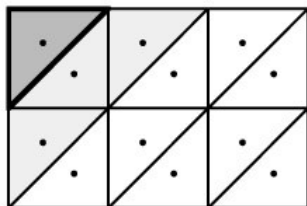
It is relatively straightforward to convert a partitioner into a hill-climbing partitioner by remembering the order that faces were selected for inclusion in partitions, perturbing that order slightly, and re-evaluating the results. By recording the best sequence, you can use that as the basis for perturbations. It is unlikely to perform significantly better than the core algorithm, but as slight as the improvements may be, if squeezing out the absolute best performance and smallest memory footprint is your objective, it may be worth doing.

13.5 Graphical Walkthrough

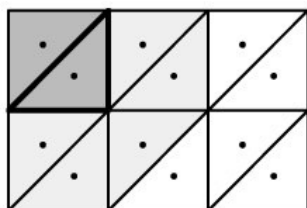
Here follows a simple graphical walkthrough that shows a partitioning into two equal parts. The diagrams show unassigned triangles, assigned triangles, and candidate triangles for inclusion in different shadings for clarity.



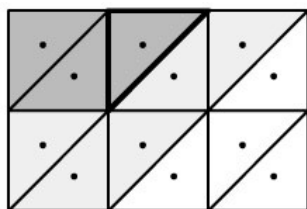
Step 1. A simple input mesh. Triangulation is not strictly necessary, but your implementation will be far simpler, and likely have fewer bugs and better performance as a result.



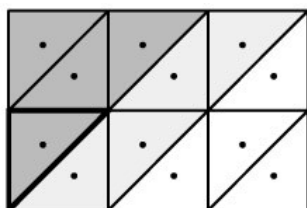
Step 2. The top-left corner is one of the least connected triangles in the mesh. We initialize the first partition with it. Next, we discover the candidate triangles that have vertices in common with the current partition. This is easily done by creating a map of shared vertices to related triangles.



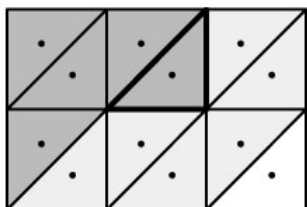
Step 3. Next, one of the candidates is selected based on minimum bounding radius, distance between the candidate triangle and its related triangle in the partition (related through the sharing over vertices, not edge connectivity), or other factors.



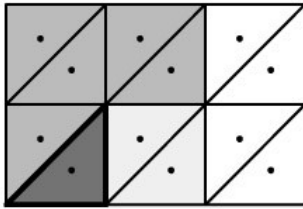
Step 4. Note how candidate triangles may not have edges in common with the current partition.



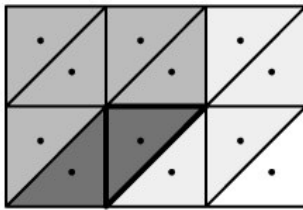
Step 5. The newly added triangle was a good choice because it minimized the bounding volume of the current partition and only added one vertex.



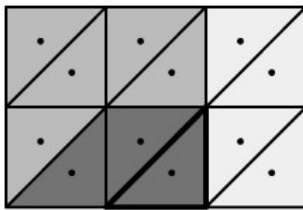
Step 6. Let's pretend that we cannot add any new triangles because some threshold has been reached. So, we search all unassigned triangles, not necessarily for one that is *near* the current partition, but for one that is *least connected to the remaining unassigned triangles* in the mesh.



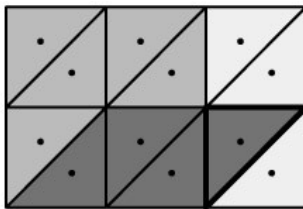
Step 7. Here, we have selected a triangle that has only two related triangles. Every other unassigned triangle is related to at least three others. Remember, relationship is determined by vertex data sharing, not edges.



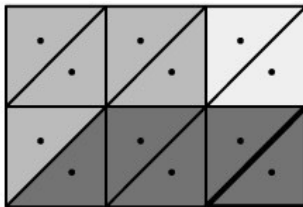
Step 8. Two vertices are shared with this triangle, so adding it only costs one new vertex.



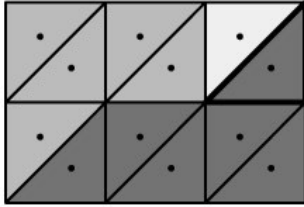
Step 9. This triangle only requires one new vertex.



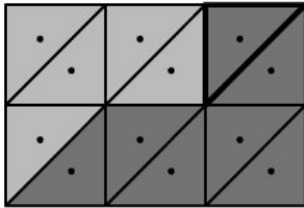
Step 10. This triangle increases the bounding sphere minimally, and only adds one new vertex.



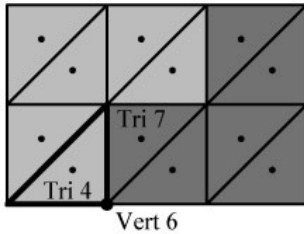
Step 11. This triangle adds one new vertex, and is closer, measured by distance between face centers, to the related triangle than the similar triangle not selected above.



Step 12. This triangle adds one new vertex.



Step 13. We've run out of triangles to assign. They aren't perfectly balanced, so let's run a refinement step to improve the balance.



Step 14. During the refinement process, both partitions are examined for triangles that would prefer to be somewhere else, at no cost. Since that situation does not occur, the only way to balance the partitions is to move Vertex 6 to the first partition along with the triangle indices for Triangle 4. Provided neither resulting partition fails to fit under their thresholds (bounding radius, vertex count, triangle count, etc.), this is done.

Note that Triangle 7 now can be moved freely between the partitions without changing the vertex count. In this case, there's no reason to do so since the partitions are balanced and vertex count cannot be reduced.

References

[1] J. B. Kruskal. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem". *Proceedings of the American Mathematical Society*, Volume 7, Number 1 (February, 1956), pp. 48–50.

- [2] R. C. Prim. "Shortest connection networks and some generalizations". *Bell System Technical Journal*, Volume 36 (1957), pp. 1389–1401.
- [3] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. "MultiChart Geometry Images". *Proceedings of the 2003 Eurographics Symposium on Geometry Processing*, 2003.

14

Moments of Inertia for Common Shapes

Eric Lengyel

Terathon Software

Overview

The moment of inertia is an important quantity in rigid body dynamics. It's the rotational analog of mass, and it describes how difficult it is to change the angular velocity of an object. The formula used to calculate the moment of inertia I about a particular axis is the integral

$$(14.1) \quad I = \int_v r^2 dm$$

14.1 Center of Mass

In a rigid body simulation, it is most useful to know the moment of inertia for an object about its center of mass because that is the point about which the object naturally rotates. In order to calculate the center of mass for an object, we first need to be able to calculate the object's mass. If we consider an object to be composed of a large number of particles, then its total mass m is simply the sum of the masses m_k of those particles:

$$(14.2) \quad m = \sum_k m_k .$$

The center of mass \mathbf{C} is found by taking the product of each particle's mass m_k and its position \mathbf{r}_k , summing over all particles, and then dividing by the total mass as follows:

$$(14.3) \quad \mathbf{C} = \frac{\sum_k \mathbf{r}_k m_k}{\sum_k m_k} .$$

For a continuous volume, these summations become integrals. The mass m of an object is found by integrating the object's density over its volume as follows:

$$(14.4) \quad m = \int_v \rho(\mathbf{r}) dv .$$

Here, $\rho(\mathbf{r})$ is a function that gives the density of the object at any point \mathbf{r} inside its volume, and $dv = dx \, dy \, dz$ is a differential volume element. The density is often a constant that we can move out of the integral, so we drop the function notation and simply write it as ρ :

$$(14.5) \quad m = \rho \int_v dv .$$

The center of mass \mathbf{C} for an object is found by integrating the product of the differential mass and its position and dividing the result by the total mass as follows:

$$(14.6) \quad \mathbf{C} = \frac{\rho \int_v \mathbf{r} dv}{\rho \int_v dv} .$$

14.2 The Inertia Tensor

In a given coordinate system, every rigid body has three moments of inertia (one for each of the coordinate axes) and three products of inertia. These six quantities form what is called the *inertia tensor* for the rigid body. The inertia tensor is ordinarily expressed as a 3×3 matrix, but it is symmetric, so there are only six distinct entries. For a set of particles, the inertia tensor \mathbf{I} is given by the formula

$$(14.7) \quad \mathbf{I} = \sum_k m_k \begin{bmatrix} y_k^2 + z_k^2 & -x_k y_k & -x_k z_k \\ -x_k y_k & x_k^2 + z_k^2 & -y_k z_k \\ -x_k z_k & -y_k z_k & x_k^2 + y_k^2 \end{bmatrix}.$$

where the k -th particle has mass m_k and is located at the point (x_k, y_k, z_k) [1]. This formula can also be expressed as

$$(14.8) \quad \mathbf{I} = \sum_k m_k (r_k^2 \mathbf{E}_3 - \mathbf{r}_k \otimes \mathbf{r}_k),$$

where \mathbf{E}_3 is the 3×3 identity matrix, $\mathbf{r}_k = (x_k, y_k, z_k)$, and the operation \otimes is the tensor product giving

$$(14.9) \quad \mathbf{r}_k \otimes \mathbf{r}_k = \begin{bmatrix} x_k^2 & x_k y_k & x_k z_k \\ x_k y_k & y_k^2 & y_k z_k \\ x_k z_k & y_k z_k & z_k^2 \end{bmatrix}.$$

The diagonal entries of the inertia tensor are the moments of inertia, and the off-diagonal entries are the products of inertia. It is always possible to find a coordinate system in which the products of inertia are all zero, and we call the axes of such a

coordinate system the *principal axes of inertia* for a rigid body. In this gem, we only compute the inertia tensor in a coordinate system aligned to the principal axes. The orientation of these axes are usually evident due to symmetry in the object being examined.

For a continuous volume in a coordinate system aligned to the principal axes of inertia, the diagonal entries of the inertia tensor \mathbf{I} are given by the integrals

$$\begin{aligned}
 I_{11} &= \rho \int_v (y^2 + z^2) dv \\
 (14.10) \quad I_{22} &= \rho \int_v (x^2 + z^2) dv \\
 I_{33} &= \rho \int_v (x^2 + y^2) dv
 \end{aligned}$$

Transformations

Given an invertible 3×3 transformation matrix \mathbf{M} that transforms points from one coordinate system to another coordinate system with the same origin, an inertia tensor \mathbf{I} is transformed according to the formula

$$(14.11) \quad \mathbf{I}' = \mathbf{M} \mathbf{I} \mathbf{M}^{-1} .$$

It's useful to think of this product as first transforming in reverse from the new coordinate system to the original coordinate system using \mathbf{M}^{-1} , applying the inertia tensor \mathbf{I} in that coordinate system, and then transforming back into the new coordinate system using \mathbf{M} .

To transform an inertia tensor into a coordinate system with a different origin, we can use a formula known as the *parallel axis theorem*. Let \mathbf{s} be an offset vector

representing the difference between the new origin and the old origin. Then, starting with the formula for the inertia tensor given in Equation (14.8), we replace \mathbf{r} with $\mathbf{r}+\mathbf{s}$ to obtain

$$(14.12) \quad \mathbf{I}' = \sum_k m_k [(\mathbf{r}_k + \mathbf{s})^2 \mathbf{E}_3 - (\mathbf{r}_k + \mathbf{s}) \otimes (\mathbf{r}_k + \mathbf{s})] .$$

Expanding this summation, we have

$$(14.13) \quad \begin{aligned} \mathbf{I}' = \sum_k m_k r_k^2 \mathbf{E}_3 &+ \left[2 \left(\sum_k m_k \mathbf{r}_k \right) \cdot \mathbf{s} \right] \mathbf{E}_3 + \left(\sum_k m_k \right) s^2 \mathbf{E}_3 \\ &- \sum_k m_k \mathbf{r}_k \otimes \mathbf{r}_k - \left(\sum_k m_k \mathbf{r}_k \right) \otimes \mathbf{s} - \mathbf{s} \otimes \sum_k m_k \mathbf{r}_k \\ &- \left(\sum_k m_k \right) \mathbf{s} \otimes \mathbf{s} \end{aligned}$$

This equation contains the two terms from original summation given by Equation (14.8) for \mathbf{I} , so we can substitute \mathbf{I} for these terms to get

$$(14.14) \quad \begin{aligned} \mathbf{I}' = \mathbf{I} &+ \left[2 \left(\sum_k m_k \mathbf{r}_k \right) \cdot \mathbf{s} \right] \mathbf{E}_3 + \left(\sum_k m_k \right) s^2 \mathbf{E}_3 - \left(\sum_k m_k \mathbf{r}_k \right) \otimes \mathbf{s} \\ &- \mathbf{s} \otimes \sum_k m_k \mathbf{r}_k - \left(\sum_k m_k \right) \mathbf{s} \otimes \mathbf{s} . \end{aligned}$$

Now, if the origin of the coordinate system coincides with the center of mass, then the summation $\sum_k m_k \mathbf{r}_k$ is equal to the point (0,0,0). This allows us to make a tremendous simplification because all of the terms in Equation (14.14) containing this summation

vanish. We therefore can use the formula

$$(14.15) \quad \mathbf{I}' = \mathbf{I} + m(s^2 \mathbf{E}_3 - \mathbf{s} \otimes \mathbf{s}).$$

to transform an inertia tensor from a coordinate system in which the center of mass lies at the origin to another coordinate system in which the new origin lies at the point \mathbf{s} in the original coordinate system.

It's important to understand that Equation (14.15) can only be applied once to an inertia tensor in order to move it away from the center of mass. After the inertia tensor has been moved, it no longer uses a coordinate system in which the origin coincides with the center of mass, but that condition must be true for Equation (14.15) to be valid. However, it is possible to recover the inertia tensor \mathbf{I} from the offset inertia tensor \mathbf{I}' if the vector \mathbf{s} is known, once again

14.3 Derivation of Moments of Inertia

In this section, we derive the centers of mass and the moments of inertia for a variety of common solid shapes. The inertia tensors are always expressed in a coordinate system in which the origin lies at the center of mass and the coordinate axes are parallel to the shape's principal axes of inertia.

Evaluating integrals of the type presented in this section by hand can be a very tedious exercise. We recommend using a symbolic computation package such as *Mathematica* to perform these calculations, should the reader feel so inclined.

Box

There are two common ways to describe the dimensions of a box, as shown in Figure 14.1. One way is to place the origin at one corner and identify the full extents of the box in all three directions by its length l , its width w , and its height h . The second

way is to place the origin at the center of the box and identify the perpendicular distances a , b , and c from the center to the faces in all three directions. We provide formulas for both cases.

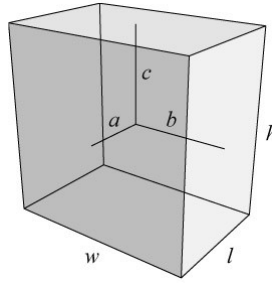


Figure 14.1: A box.

In the case that the box is described by the dimensions l , w , and h , the total mass is $m = plwh$, and the center of mass is located at $(l/2, w/2, h/2)$. The moments of inertia are then given by the integrals

$$\begin{aligned}
 I_{11} &= \rho \int_{-h/2}^{h/2} \int_{-w/2}^{w/2} \int_{-l/2}^{l/2} (y^2 + z^2) dx dy dz = \frac{1}{12} \rho lwh (w^2 + h^2), \\
 (14.16) \quad I_{22} &= \rho \int_{-h/2}^{h/2} \int_{-w/2}^{w/2} \int_{-l/2}^{l/2} (x^2 + z^2) dx dy dz = \frac{1}{12} \rho lwh (l^2 + h^2), \\
 I_{33} &= \rho \int_{-h/2}^{h/2} \int_{-w/2}^{w/2} \int_{-l/2}^{l/2} (x^2 + y^2) dx dy dz = \frac{1}{12} \rho lwh (l^2 + w^2).
 \end{aligned}$$

Substituting the mass m , this gives us the inertia tensor

$$(14.17) \quad \mathbf{I}_{\text{box}} = \begin{bmatrix} \frac{1}{12}m(w^2 + h^2) & 0 & 0 \\ 0 & \frac{1}{12}m(l^2 + h^2) & 0 \\ 0 & 0 & \frac{1}{12}m(l^2 + w^2) \end{bmatrix}.$$

In the case that the box is described by the dimensions a , b , and c , the total mass is $m = 8\rho abc$, and the center of mass coincides with the origin. The moments of inertia are then given by the integrals

$$(14.18) \quad \begin{aligned} I_{11} &= \rho \int_{-c}^c \int_{-b}^b \int_{-a}^a (y^2 + z^2) dx dy dz = \frac{8}{3} \rho abc (b^2 + c^2), \\ I_{22} &= \rho \int_{-c}^c \int_{-b}^b \int_{-a}^a (x^2 + z^2) dx dy dz = \frac{8}{3} \rho abc (a^2 + c^2), \\ I_{33} &= \rho \int_{-c}^c \int_{-b}^b \int_{-a}^a (x^2 + y^2) dx dy dz = \frac{8}{3} \rho abc (a^2 + b^2). \end{aligned}$$

Substituting the mass m , this gives us the inertia tensor

$$(14.19) \quad \mathbf{I}_{\text{box}} = \begin{bmatrix} \frac{1}{3}m(b^2 + c^2) & 0 & 0 \\ 0 & \frac{1}{3}m(a^2 + c^2) & 0 \\ 0 & 0 & \frac{1}{3}m(a^2 + b^2) \end{bmatrix}.$$

Cylinder

The dimensions of a cylinder are described by its height h and the two semi-axis lengths a and b of its base, as shown in Figure 14.2. If the cylinder is circular, then $a = b$.

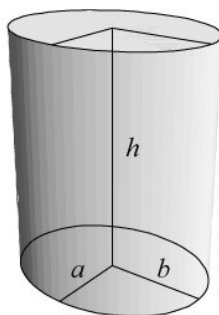


Figure 14.2: A cylinder.

The center of mass lies at the point $(0, 0, h/2)$, and the total mass of the cylinder is $m = \rho\pi abh$. The moments of inertia are then given by the integrals

$$\begin{aligned}
 I_{11} &= \rho \int_{-h/2}^{h/2} \int_{-b}^b \int_{-a\sqrt{1-y^2/b^2}}^{a\sqrt{1-y^2/b^2}} (y^2 + z^2) dx dy dz = \pi\rho abh \left(\frac{1}{4}b^2 + \frac{1}{12}h^2 \right), \\
 (14.20) \quad I_{22} &= \rho \int_{-h/2}^{h/2} \int_{-b}^b \int_{-a\sqrt{1-y^2/b^2}}^{a\sqrt{1-y^2/b^2}} (x^2 + z^2) dx dy dz = \pi\rho abh \left(\frac{1}{4}a^2 + \frac{1}{12}h^2 \right), \\
 I_{33} &= \rho \int_{-h/2}^{h/2} \int_{-b}^b \int_{-a\sqrt{1-y^2/b^2}}^{a\sqrt{1-y^2/b^2}} (x^2 + y^2) dx dy dz = \frac{\pi}{4}\rho abh(a^2 + b^2).
 \end{aligned}$$

Substituting the mass m , this gives us the inertia tensor

$$(14.21) \quad \mathbf{I}_{\text{box}} = \begin{bmatrix} \frac{1}{4}mb^2 + \frac{1}{12}mh^2 & 0 & 0 \\ 0 & \frac{1}{4}ma^2 + \frac{1}{12}mh^2 & 0 \\ 0 & 0 & \frac{1}{4}m(a^2 + b^2) \end{bmatrix}.$$

Pyramid

The dimensions of a rectangular pyramid are described by its height h and the perpendicular distances a_0 and b_0 from the center of the base to two adjacent edges of the base, as shown in Figure 14.3.

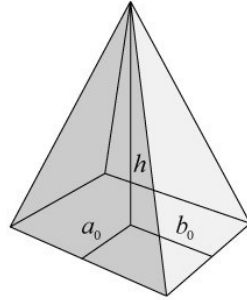


Figure 14.3: A rectangular pyramid.

In order to calculate the total mass and center of mass, we need to be able to express the lengths a and b of a cross-section of the pyramid at any z -coordinate. Functions $a(z)$ and $b(z)$ producing the base lengths at $z = 0$ and linearly tapering to zero at the apex where $z = h$ are given by

$$(14.22) \quad \begin{aligned} a(z) &= a_0 \frac{h-z}{h} \\ b(z) &= b_0 \frac{h-z}{h}. \end{aligned}$$

The total mass is then given by integrating rectangular areas over the entire height of the pyramid:

$$(14.23) \quad m = 4\rho \int_0^h a(z)b(z)dz = 4\rho \frac{a_0 b_0}{h^2} \int_0^h (h-z)^2 dz = \frac{4}{3} \rho a_0 b_0 h.$$

The center of mass clearly lies on the z -axis, and we can calculate its z -coordinate by multiplying a factor of z into the integrand for the mass to obtain

$$(14.24) \quad mC_z = 4\rho \frac{a_0 b_0}{h^2} \int_0^h (h-z)^2 z dz = \frac{1}{3} \rho a_0 b_0 h^2.$$

After dividing by m , we find the center of mass to be located at the point $(0,0, h/4)$.

Since the moment of inertia is best calculated in the coordinate system for which the origin coincides with the center of mass, it is useful to consider a pyramid that extends from $-h/4$ to $3h/4$ in the z direction and redefine the functions $a(z)$ and $b(z)$ as

$$(14.25) \quad \begin{aligned} a(z) &= \frac{a_0}{h} \left(\frac{3}{4}h - z \right) \\ b(z) &= \frac{b_0}{h} \left(\frac{3}{4}h - z \right) \end{aligned}$$

The moments of inertia are then given by the integrals

$$\begin{aligned}
 I_{11} &= \rho \int_{-\frac{h}{4}}^{\frac{3h}{4}} \int_{-b(z)}^{b(z)} \int_{-a(z)}^{a(z)} (y^2 + z^2) dx dy dz = \rho a_0 b_0 h \left(\frac{4}{15} b_0^2 + \frac{1}{20} h^2 \right), \\
 (14.26) \quad I_{22} &= \rho \int_{-h/4}^{3h/4} \int_{-b(z)}^{b(z)} \int_{-a(z)}^{a(z)} (x^2 + z^2) dx dy dz = \rho a_0 b_0 h \left(\frac{4}{15} a_0^2 + \frac{1}{20} h^2 \right), \\
 I_{33} &= \rho \int_{-h/4}^{3h/4} \int_{-b(z)}^{b(z)} \int_{-a(z)}^{a(z)} (x^2 + y^2) dx dy dz = \frac{4}{15} \rho a_0 b_0 h (a_0^2 + b_0^2),
 \end{aligned}$$

Substituting the mass m , this gives us the inertia tensor

$$(14.27) \quad \mathbf{I}_{\text{pyramid}} = \begin{bmatrix} \frac{1}{5} m b_0^2 + \frac{3}{80} m h^2 & 0 & 0 \\ 0 & \frac{1}{5} m a_0^2 + \frac{3}{80} m h^2 & 0 \\ 0 & 0 & \frac{1}{5} m (a_0^2 + b_0^2) \end{bmatrix}.$$

Cone

The dimensions of a cone are described by its height h and the two semi-axis lengths a_0 and b_0 of its base, as shown in Figure 14.4. If the cone is circular, then $a_0 = b_0$.

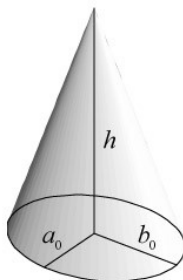


Figure 14.4: A cone.

As with the pyramid, we use the functions $a(z)$ and $b(z)$ given by Equation (14.22) to express the dimensions of a cross-section of the cone at a height z above the base. We can calculate the total mass of the cone by integrating elliptical disk areas over the entire height of the cone:

$$(14.28) \quad m = \rho\pi \int_0^h a(z)b(z)dz = \rho\pi \frac{a_0b_0}{h^2} \int_0^h (h-z)^2 dz = \frac{\pi}{3} \rho a_0 b_0 h.$$

The center of mass clearly lies on the z -axis, and we can calculate its z -coordinate by multiplying a factor of z into the integrand for the mass to obtain

$$(14.29) \quad mC_z = \rho\pi \frac{a_0b_0}{h^2} \int_0^h (h-z)^2 z dz = \frac{\pi}{12} \rho a_0 b_0 h^2.$$

After dividing by m , we find the center of mass to be located at the point $(0,0,h/4)$.

It is no coincidence that the centers of mass for the pyramid and cone are equal. The same point is obtained for any two-dimensional base shape that linearly tapers to a point at a height h . To calculate the moments of inertia in a coordinate system having the origin at the center of mass, we again redefine the functions $a(z)$ and $b(z)$ as in

Equation (14.25) and integrate from $-h/4$ to $3h/4$. The moments of inertia are then given by the integrals

$$\begin{aligned}
 I_{11} &= \rho \int_{-h/4}^{3h/4} \int_{-b(z)}^{b(z)} \int_{-a(z)\sqrt{1-y^2/b(z)^2}}^{a(z)\sqrt{1-y^2/b(z)^2}} (y^2 + z^2) dx dy dz \\
 &= \pi \rho a_0 b_0 h \left(\frac{1}{20} b_0^2 + \frac{1}{80} h^2 \right), \\
 I_{22} &= \rho \int_{-\frac{h}{4}}^{\frac{3h}{4}} \int_{-b(z)}^{b(z)} \int_{-a(z)\sqrt{1-y^2/b(z)^2}}^{a(z)\sqrt{1-y^2/b(z)^2}} (x^2 + z^2) dx dy dz \\
 (14.30) \quad &= \pi \rho a_0 b_0 h \left(\frac{1}{20} a_0^2 + \frac{1}{80} h^2 \right), \\
 I_{33} &= \rho \int_{-h/4}^{3h/4} \int_{-b(z)}^{b(z)} \int_{-a(z)\sqrt{1-y^2/b(z)^2}}^{a(z)\sqrt{1-y^2/b(z)^2}} (x^2 + y^2) dx dy dz \\
 &= \frac{\pi}{20} \rho a_0 b_0 h (a_0^2 + b_0^2).
 \end{aligned}$$

Substituting the mass m , this gives us the inertia tensor

$$(14.31) \quad \mathbf{I}_{\text{cone}} = \begin{bmatrix} \frac{3}{20} m b_0^2 + \frac{3}{80} m h^2 & 0 & 0 \\ 0 & \frac{3}{20} m a_0^2 + \frac{3}{80} m h^2 & 0 \\ 0 & 0 & \frac{3}{20} m (a_0^2 + b_0^2) \end{bmatrix}.$$

Ellipsoid

The dimensions of an ellipsoid are described by the three semi-axis lengths a , b , and c , as shown in Figure 14.5. In the case of a sphere, $a=b=c$. The center of mass is clearly located at the center of the ellipsoid, and that is where we place the origin as well. The total mass of the ellipsoid is $m =$

$$\frac{4}{3}\pi\rho abc$$

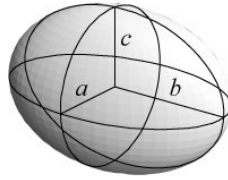


Figure 14.5: An ellipsoid.

To make the integrals simpler, we remap an ellipsoid to a sphere of radius one using the following substitutions:

$$(14.32) \quad \begin{aligned} u &= \frac{x}{a} & du &= \frac{1}{a} dx \\ u &= \frac{y}{b} & dv &= \frac{1}{b} dy \\ u &= \frac{z}{c} & dw &= \frac{1}{c} dz \end{aligned}$$

The moments of inertia for an ellipsoid are then given by the integrals

$$(14.33) \quad \begin{aligned} I_{11} &= \rho abc \int_{-1}^1 \int_{-\sqrt{1-w^2}}^{\sqrt{1-w^2}} \int_{-\sqrt{1-v^2-w^2}}^{\sqrt{1-v^2-w^2}} (b^2 v^2 + c^2 w^2) du \, dv \, dw \\ &= \frac{4\pi}{15} \rho abc (b^2 + c^2), \end{aligned}$$

$$\begin{aligned}
I_{22} &= \rho abc \int_{-1}^1 \int_{-\sqrt{1-w^2}}^{\sqrt{1-w^2}} \int_{-\sqrt{1-v^2-w^2}}^{\sqrt{1-v^2-w^2}} (a^2 u^2 + c^2 w^2) du dv dw \\
&= \frac{4\pi}{15} \rho abc (a^2 + c^2), \\
I_{33} &= \rho abc \int_{-1}^1 \int_{-\sqrt{1-w^2}}^{\sqrt{1-w^2}} \int_{-\sqrt{1-v^2-w^2}}^{\sqrt{1-v^2-w^2}} (a^2 u^2 + b^2 v^2) du dv dw \\
&= \frac{4\pi}{15} \rho abc (a^2 + b^2).
\end{aligned}$$

Substituting the mass m , this gives us the inertia tensor

$$(14.34) \quad \mathbf{I}_{\text{ellipsoid}} = \begin{bmatrix} \frac{1}{5}m(b^2 + c^2) & 0 & 0 \\ 0 & \frac{1}{5}m(a^2 + c^2) & 0 \\ 0 & 0 & \frac{1}{5}m(a^2 + b^2) \end{bmatrix}.$$

Dome

The dimensions of a dome, or ellipsoidal hemisphere, are described in the same way as a complete ellipsoid: by the semi-axis lengths a , b , and c , as shown in Figure 14.6.

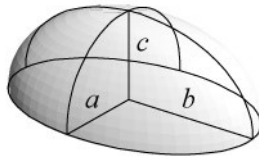


Figure 14.6: A dome, or ellipsoidal hemisphere.

The total mass of a dome is $m = \frac{2}{3}\pi abc$, and the z -coordinate of the center of mass can be calculated using the integral

$$(14.21) \quad mC_z = \rho abc^2 \int_0^1 \int_{-\sqrt{1-w^2}}^{\sqrt{1-w^2}} \int_{-\sqrt{1-v^2-w^2}}^{\sqrt{1-v^2-w^2}} w \, du \, dv \, dw = \frac{\pi}{4} abc^2,$$

where we have again made the substitutions given in Equation (14.32). After dividing by m , we find the center of mass to be located at the point $(0,0,3c/8)$.

To calculate the moments of inertia for a dome, we can use a trick that makes the integrals simpler. Instead of calculating the inertia tensor \mathbf{I} about the center of mass, we calculate the inertia tensor \mathbf{I}' about the origin at the center of the dome's base and then use Equation (14.15) to find \mathbf{I} when the offset is $\mathbf{s}=(0,0,-3c/8)$. The moments of inertia about the origin for a dome are given by the integrals

$$(14.36) \quad \begin{aligned} I_{11} &= \rho \int_{-h/2}^{h/2} \int_{-w/2}^{w/2} \int_{-l/2}^{l/2} (y^2 + z^2) dx \, dy \, dz = \frac{1}{12} \rho l w h (w^2 + h^2), \\ I_{22} &= \rho \int_{-h/2}^{h/2} \int_{-w/2}^{w/2} \int_{-l/2}^{l/2} (x^2 + z^2) dx \, dy \, dz = \frac{1}{12} \rho l w h (l^2 + h^2), \\ I_{33} &= \rho \int_{-h/2}^{h/2} \int_{-w/2}^{w/2} \int_{-l/2}^{l/2} (x^2 + y^2) dx \, dy \, dz = \frac{1}{12} \rho l w h (l^2 + w^2). \end{aligned}$$

Substituting the mass m , this gives us the inertia tensor

$$(14.37) \quad \mathbf{I}'_{\text{dome}} = \begin{bmatrix} \frac{1}{5}m(b^2 + c^2) & 0 & 0 \\ 0 & \frac{1}{5}m(a^2 + c^2) & 0 \\ 0 & 0 & \frac{1}{5}m(a^2 + b^2) \end{bmatrix}.$$

This is identical to the inertia tensor for an ellipsoid, but the mass m has been cut in half. In order to obtain the inertia tensor \mathbf{I}_{dome} about the center of mass, we must calculate

$$(14.38) \quad \mathbf{I}_{\text{dome}} = \mathbf{I}'_{\text{dome}} + m(s^2\mathbf{E}_3 - \mathbf{s} \otimes \mathbf{s}).$$

With $\mathbf{s}=(0,0,-3c/8)$, we have

$$(14.39) \quad s^2\mathbf{E}_3 - \mathbf{s} \otimes \mathbf{s} = \begin{bmatrix} \frac{9}{64}c^2 & 0 & 0 \\ 0 & \frac{9}{64}c^2 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

and so

$$(14.40) \quad \mathbf{I}_{\text{dome}} = \begin{bmatrix} \frac{1}{5}mb^2 + \frac{19}{320}mc^2 & 0 & 0 \\ 0 & \frac{1}{5}ma^2 + \frac{19}{320}mc^2 & 0 \\ 0 & 0 & \frac{1}{5}m(a^2 + b^2) \end{bmatrix}.$$

Capsule

The dimensions of a capsule are described by the height h of a central cylinder, the

two semi-axis lengths a and b of the cylinder's base, and a third semi-axis length c representing the extent of each hemispherical end cap in the direction perpendicular to the cylinder's base, as shown in Figure 14.7.

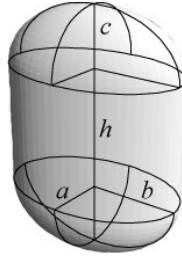


Figure 14.7: A capsule.

The total mass m_{capsule} is given by the sum

$$(14.41) \quad m_{\text{capsule}} = m_{\text{cylinder}} + 2m_{\text{dome}} ,$$

where $m_{\text{cylinder}} = \rho\pi abh$ is the mass of the central cylinder, and $m_{\text{dome}} = \frac{2}{3}\rho\pi abc$ is the mass of a single hemispherical end cap. Due to symmetry, it is clear that the center of mass lies at the center of the cylindrical portion of the capsule.

We can calculate the inertia tensor capsule $\mathbf{I}_{\text{capsule}}$ by combining the inertia tensors of the cylinder and dome in the proper manner. With respect to an origin located at the capsule's center of mass, the centers of mass for the hemispherical end caps lie at the z -coordinates

$$(14.42) \quad z_{\text{cap}} = \pm \left(\frac{h}{2} + \frac{3}{8}c \right) .$$

Using the offset formula given by Equation (14.15) with $\mathbf{s} = (0, 0, z_{\text{cap}})$, we can transform the inertia tensor for a dome into the capsule's coordinate system to obtain

$$(14.43) \quad \mathbf{I}_{\text{cap}} = \mathbf{I}_{\text{dome}} + m_{\text{dome}} \begin{bmatrix} \frac{9}{64}c^2 + \frac{3}{8}hc + \frac{1}{4}h^2 & 0 & 0 \\ 0 & \frac{9}{64}c^2 + \frac{3}{8}hc + \frac{1}{4}h^2 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Doubling this to account for both end caps and adding it to the inertia tensor for a cylinder gives us

$$(14.44) \quad \begin{aligned} (\mathbf{I}_{\text{capsule}})_{11} &= \frac{2}{5}m_{\text{dome}}(b^2 + c^2) + \frac{3}{4}m_{\text{dome}}hc + \frac{1}{2}m_{\text{dome}}h^2 \\ &\quad + \frac{1}{4}m_{\text{cylinder}}b^2 + \frac{1}{12}m_{\text{cylinder}}h^2 \\ (\mathbf{I}_{\text{capsule}})_{22} &= \frac{2}{5}m_{\text{dome}}(a^2 + c^2) + \frac{3}{4}m_{\text{dome}}hc + \frac{1}{2}m_{\text{dome}}h^2 \\ &\quad + \frac{1}{4}m_{\text{cylinder}}a^2 + \frac{1}{12}m_{\text{cylinder}}h^2 \\ (\mathbf{I}_{\text{capsule}})_{33} &= \frac{2}{5}m_{\text{dome}}(a^2 + b^2) + \frac{1}{4}m_{\text{cylinder}}(a^2 + b^2). \end{aligned}$$

These moments of inertia are given in terms of the overall mass of the capsule in the summary at the end of this gem.

Truncated Pyramid

A truncated pyramid is a pyramid that has been cut off at some height h above the base by a plane parallel to the base. As with a pyramid, we describe the dimensions of the base by the perpendicular distances a_0 and b_0 from the center of the base to two adjacent edges of the base, as shown in Figure 14.8. We introduce a factor r representing the ratio of the length of an edge on the top face to the length of the corresponding edge

on the bottom face (the base). The dimensions of the top face are then described by the perpendicular distances ra_0 and rb_0 from the center to the edges. Where $r = 0$, all of the formulas for a truncated pyramid reduce to those for a complete pyramid.

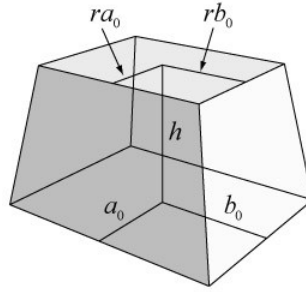


Figure 14.8: A truncated pyramid.

In order to calculate the total mass and center of mass, we express the lengths a and b as functions of the z -coordinate as follows:

$$(14.45) \quad \begin{aligned} a(z) &= a_0 \left(1 + \frac{r-1}{h} z \right) \\ b(z) &= b_0 \left(1 + \frac{r-1}{h} z \right). \end{aligned}$$

The total mass is then given by integrating rectangular areas over the range of z -coordinates between the bottom and top faces of the truncated pyramid:

$$(14.46) \quad m = 4\rho \int_0^h a(z)b(z)dz = \frac{4}{3}\rho a_0 b_0 h(r^2 + r + 1).$$

The center of mass clearly lies on the z -axis, and we can calculate its z -coordinate by multiplying a factor of z into the integrand for the mass to obtain

$$(14.47) \quad mC_z = 4\rho \int_0^h a(z)b(z)z \, dz = \frac{1}{3}\rho a_0 b_0 h^2(3r^2 + 2r + 1).$$

After dividing by m , we find the z -coordinate of the center of mass to be located at

$$(14.48) \quad C_z = \frac{3r^2 + 2r + 1}{4(r^2 + r + 1)}h.$$

We place the origin at the center of mass by shifting the range of z -coordinates downward by C_z and adding this shift back when evaluating the functions $a(z)$ and $b(z)$. The moments of inertia are then given by the integrals

$$(14.49) \quad \begin{aligned} I_{11} &= \rho \int_{-C_z}^{h-C_z} \int_{-b(z+C_z)}^{b(z+C_z)} \int_{-a(z+C_z)}^{a(z+C_z)} (y^2 + z^2) dx \, dy \, dz \\ &= \frac{4}{15}\rho a_0 b_0^3 h(r^4 + r^3 + r^2 + r + 1) \\ &\quad + \rho a_0 b_0 h^3 \frac{r^4 + 4r^3 + 10r^2 + 4r + 1}{20(r^2 + r + 1)}, \end{aligned}$$

$$\begin{aligned} I_{22} &= \rho \int_{-C_z}^{h-C_z} \int_{-b(z+C_z)}^{b(z+C_z)} \int_{-a(z+C_z)}^{a(z+C_z)} (x^2 + z^2) dx \, dy \, dz \\ &= \frac{4}{15}\rho a_0^3 b_0 h(r^4 + r^3 + r^2 + r + 1) \\ &\quad + \rho a_0 b_0 h^3 \frac{r^4 + 4r^3 + 10r^2 + 4r + 1}{20(r^2 + r + 1)}, \end{aligned}$$

$$\begin{aligned}
I_{33} &= \rho \int_{-C_z}^{h-C_z} \int_{-b(z+C_z)}^{b(z+C_z)} \int_{-a(z+C_z)}^{a(z+C_z)} (x^2 + y^2) dx dy dz \\
&= \frac{4}{15} \rho a_0 b_0 h (a_0^2 + b_0^2) (r^4 + r^3 + r^2 + r + 1).
\end{aligned}$$

Substituting the mass m , this gives us the inertia tensor

$$\begin{aligned}
&(\mathbf{I}_{\text{trunc-pyramid}})_{11} \\
&= \frac{1}{5} m b_0^2 \frac{r^4 + r^3 + r^2 + r + 1}{r^2 + r + 1} \\
&\quad + \frac{1}{80} m h^2 \frac{r^4 + 4r^3 + 10r^2 + 4r + 1}{(r^2 + r + 1)^2} \\
(14.50) \quad &(\mathbf{I}_{\text{trunc-pyramid}})_{22} \\
&= \frac{1}{5} m a_0^2 \frac{r^4 + r^3 + r^2 + r + 1}{r^2 + r + 1} \\
&\quad + \frac{1}{80} m h^2 \frac{r^4 + 4r^3 + 10r^2 + 4r + 1}{(r^2 + r + 1)^2} \\
&(\mathbf{I}_{\text{trunc-pyramid}})_{33} = \frac{1}{5} m (a_0^2 + b_0^2) \frac{r^4 + r^3 + r^2 + r + 1}{r^2 + r + 1}.
\end{aligned}$$

Truncated Cone

A truncated cone is a cone that has been cut off at some height h above the base by a plane parallel to the base. As with a cone, we describe the dimensions of the base by the semi-axis lengths a_0 and b_0 , as shown in Figure 14.9. We introduce a factor r representing the ratio of a semi-axis length of the top face to the corresponding semi-axis length of the bottom face (the base). The dimensions of the top face are then described by the semi-axis lengths ra_0 and rb_0 . When $r = 0$, all of the formulas for a

truncated cone reduce to those for a complete cone.

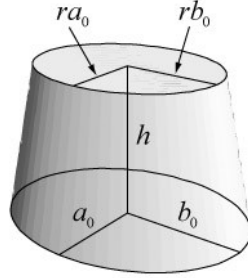


Figure 14.9: A truncated cone.

In order to calculate the total mass and center of mass, we express the lengths a and b as functions of the z -coordinate using the same formulas given by Equation (14.45) for the truncated pyramid. The total mass is then given by integrating elliptical disk areas over the range of z -coordinates between the bottom and top faces of the truncated cone:

$$(14.51) \quad m = \pi\rho \int_0^h a(z)b(z)dz = \frac{\pi}{3}\rho a_0 b_0 h(r^2 + r + 1) \quad .$$

We can then calculate the z -coordinate of the center of mass by multiplying a factor of z into the integrand for the mass to obtain

$$(14.52) \quad mC_z = \pi\rho \int_0^h a(z)b(z)zdz = \frac{\pi}{12}\rho a_0 b_0 h^2(3r^2 + 2r + 1) \quad .$$

After dividing by m , we find the z -coordinate of the center of mass to be located at

$$(14.53) \quad C_z = \frac{3r^2 + 2r + 1}{4(r^2 + r + 1)} h.$$

just as it is for the truncated pyramid. The moments of inertia are then given by the integrals

$$\begin{aligned}
 I_{11} &= \rho \int_{-C_z}^{h-C_z} \int_{-b(z+C_z)}^{b(z+C_z)} \int_{-a(z+C_z)\sqrt{1-y^2/b(z+C_z)^2}}^{a(z+C_z)\sqrt{1-y^2/b(z+C_z)^2}} (y^2 + z^2) dx dy dz \\
 &= \frac{\pi}{20} \rho a_0 b_0^3 h (r^4 + r^3 + r^2 + r + 1) \\
 &\quad + \rho \pi a_0 b_0 h^3 \frac{r^4 + 4r^3 + 10r^2 + 4r + 1}{80(r^2 + r + 1)}, \\
 (14.54) \quad I_{22} &= \rho \int_{-C_z}^{h-C_z} \int_{-b(z+C_z)}^{b(z+C_z)} \int_{-a(z+C_z)\sqrt{1-y^2/b(z+C_z)^2}}^{a(z+C_z)\sqrt{1-y^2/b(z+C_z)^2}} (x^2 + z^2) dx dy dz \\
 &= \frac{\pi}{20} \rho a_0 b_0^3 h (r^4 + r^3 + r^2 + r + 1) \\
 &\quad + \rho \pi a_0 b_0 h^3 \frac{r^4 + 4r^3 + 10r^2 + 4r + 1}{80(r^2 + r + 1)}, \\
 I_{33} &= \rho \int_{-C_z}^{h-C_z} \int_{-b(z+C_z)}^{b(z+C_z)} \int_{-a(z+C_z)\sqrt{1-y^2/b(z+C_z)^2}}^{a(z+C_z)\sqrt{1-y^2/b(z+C_z)^2}} (x^2 + y^2) dx dy dz \\
 &= \frac{\pi}{20} \rho a_0 b_0 h (a_0^2 + b_0^2) (r^4 + r^3 + r^2 + r + 1).
 \end{aligned}$$

Substituting the mass m , this gives us the inertia tensor

$$\begin{aligned}
 & (\mathbf{I}_{\text{trunc-cone}})_{11} \\
 &= \frac{3}{20} m b_0^2 \frac{r^4 + r^3 + r^2 + r + 1}{r^2 + r + 1} \\
 &+ \frac{1}{80} m h^2 \frac{r^4 + 4r^3 + 10r^2 + 4r + 1}{(r^2 + r + 1)^2} \\
 (14.55) \quad & (\mathbf{I}_{\text{trunc-cone}})_{22} \\
 &= \frac{3}{20} m a_0^2 \frac{r^4 + r^3 + r^2 + r + 1}{r^2 + r + 1} \\
 &+ \frac{1}{80} m h^2 \frac{r^4 + 4r^3 + 10r^2 + 4r + 1}{(r^2 + r + 1)^2} \\
 & (\mathbf{I}_{\text{trunc-cone}})_{33} = \frac{3}{20} m (a_0^2 + b_0^2) \frac{r^4 + r^3 + r^2 + r + 1}{r^2 + r + 1}.
 \end{aligned}$$

14.4 Summary

The mass, the center of mass, and inertia tensor for each of the shapes examined in the previous section are summarized in Table 14.1.

Table 14.1: This table lists the mass m , the center of mass (CM) C , and the entries of the inertia tensor I for a variety of solid shapes. The inertia tensor is always given in the coordinate system for which the origin coincides with the center of mass. Each shape is considered to be solid with a constant density ρ .

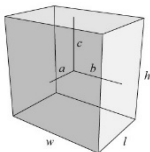
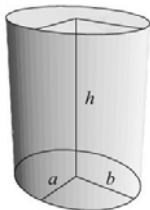
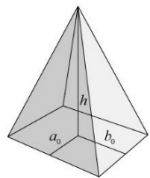
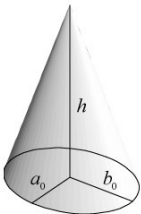


Shape	Mass and CM	Inertia Tensor
<p>Box</p> 	$m = 8\rho abc$ $C = (0,0,0)$	$I_{11} = \frac{1}{3}m(b^2 + c^2)$ $I_{22} = \frac{1}{3}m(a^2 + c^2)$ $I_{33} = \frac{1}{3}m(a^2 + b^2)$
<p>Cylinder</p> 	$m = \rho\pi abh$ $C = (0,0,\frac{h}{2})$	$I_{11} = \frac{1}{4}mb^2 + \frac{1}{12}mh^2$ $I_{22} = \frac{1}{4}ma^2 + \frac{1}{12}mh^2$ $I_{33} = \frac{1}{4}m(a^2 + b^2)$
<p>Pyramid</p> 	$m = \frac{4}{3}\rho\pi a_0 b_0 h$ $C = (0,0,\frac{h}{4})$	$I_{11} = \frac{1}{5}mb_0^2 + \frac{3}{80}mh^2$ $I_{22} = \frac{1}{5}ma_0^2 + \frac{3}{80}mh^2$ $I_{33} = \frac{1}{5}m(a_0^2 + b_0^2)$
<p>Cone</p>	$m = \frac{1}{3}\rho\pi abh$ $C = (0,0,\frac{h}{4})$	$I_{11} = \frac{3}{20}mb_0^2 + \frac{3}{80}mh^2$ $I_{22} = \frac{3}{20}ma_0^2 + \frac{3}{80}mh^2$ $I_{33} = \frac{3}{20}m(a_0^2 + b_0^2)$

Table 14.1: This table lists the mass m , the center of mass (CM) C , and the entries of the inertia tensor I for a variety of solid shapes. The inertia tensor is always given in the coordinate system for which the origin coincides with the center of mass. Each shape is considered to be solid with a constant density ρ .

Shape	Mass and CM	Inertia Tensor
		
Ellipsoid 	$m = \frac{4}{3}\rho\pi abc$ $C = (0,0,0)$	$I_{11} = \frac{1}{5}m(b^2 + c^2)$ $I_{22} = \frac{1}{5}m(a^2 + c^2)$ $I_{33} = \frac{1}{5}m(a^2 + b^2)$
Dome 	$m = \frac{2}{3}\rho\pi abc$ $C = (0,0,\frac{3c}{8})$	$I_{11} = \frac{1}{5}mb^2 + \frac{19}{320}mc^2)$ $I_{22} = \frac{1}{5}ma^2 + \frac{19}{320}mc^2$ $I_{33} = \frac{1}{5}m(a^2 + b^2)$

References

[1] Jerry B. Marion and Stephen T. Thornton.*Classical Dynamics*, 3rd edition. Sauders College Publishing, 1988.

Part II

Rendering Techniques

15

Physically-Based Outdoor Scene Lighting

Frank Kane

Sundog Software, LLC

Overview

Adventure games, role-playing games, and "serious" training and simulation games often need to render the same scene under various times of day. This gem provides a physically-based approach for generating realistic direct and diffuse ambient skylight for any given time and location, together with a tone-mapping operator to account for human perception. (See Figure 15.1.) This gives games with outdoor scenes a greater level of realism, and provides the physically accurate lighting required by training systems that might use your engine.



Figure 15.1: (See also Color Plates.) An outdoor scene with physically-based lighting at dusk (left) and at night (right). (*Images courtesy of Emergent Game Technologies and Sundog Software, LLC.*)

15.1 Positioning the Sun and Moon

Natural light comes primarily from the sun and moon, so the first step in lighting an outdoor scene is to know where to place these light sources. To do this, you will need an *ephemeris model* to compute the location of the sun and moon for a given time and location. Since this is a game engine book and not an astronomy book, we don't go into the details here, but refer you instead to the `Ephemeris` class in the code included on the accompanying CD. Understanding this class does require a few key concepts, which we describe here.

Numerical approximations of the position of astronomical objects are generally done in *ecliptic coordinates* for a given *epoch time*. Our `Ephemeris` class starts by computing the location of the sun, moon, and visible planets in ecliptic coordinates, which are just latitudes and longitudes relative to the plane defined by the path the sun takes across the sky. As a result, the ecliptic latitude of the sun is always zero. The algorithms used take as input Greenwich Mean Time expressed as the number of centuries elapsed since the year 2000; this is the epoch time for epoch 2000. Our `LocalTime` class will handle converting times in hours, minutes, and seconds for a given day to epoch centuries for you. While we are computing the location of the moon, we also compute the phase of the moon, which is important for nighttime lighting.

Ecliptic coordinates are not terribly useful for rendering, so you will need to transform the ecliptic polar coordinate system to a Cartesian coordinate system relative to your local horizon. Fortunately, this can be done with just a couple of 3×3 rotation matrices. Our code starts by taking the ecliptic coordinates of the sun or moon together with its distance from the Earth, and transforming that into a 3D vector in ecliptic space from the center of the Earth. Then, we rotate this vector into *equatorial coordinates*, which is a system defined by the plane of the Earth's equator instead of the plane of the Earth's revolution around the sun; doing this requires computing the Earth's tilt for the

simulated time. Finally, we transform the equatorial coordinates into *horizon coordinates* for the location on Earth that you wish to simulate. As a finishing touch, we also apply atmospheric refraction to the horizon coordinates, which affects the perceived location of the sun and moon as they approach the horizon.

Care must be taken that this final transformation is consistent with your engine's coordinate system conventions; if your users might define "north" and "up" as any arbitrary axis, you'll want to provide a means for them to influence this final transformation into local coordinates. The sun rising in the West instead of the East is an embarrassing bug that is very easy to slip through testing. You may also want to expose *geographic coordinates* for the sun and moon; instead of being relative to a specific location on the surface of the Earth, these coordinates are relative to the center of the Earth. Flight simulators that can cover large distances frequently use this coordinate system, and our `Ephemeris` class will compute this for you as well.

These same matrices are useful for transforming things such as star fields in the sky. It may sound like overkill, but having accurately positioned stars in your scenes could be important to someone creating, for example, a navigation training application.

All this work pays off when your engine's sun and moon rise at exactly the right time for the location being simulated. Got a game that takes place in the winter

15.2 Computing Natural Sunlight

Now that we know where the sun is in the sky, we can simulate what happens to its light as it passes through the atmosphere; this is called *atmospheric transmittance* and *atmospheric scattering*, and it is simulated by the `Spectrum` class in the included source code. Our approach uses a modified "Bird simple spectral model", named after Dr. Richard Bird who developed it at the Solar Energy Research Institute (now the

National Renewable Energy Laboratory) in 1984 [1]. It is relatively simple compared to other models, yet amazingly accurate. This code is what will turn sunlight red near sunset, for example.

Our `Spectrum` class operates over a full visible spectrum, and starts with data from NASA on the spectrum of the sun from outside of the atmosphere as input. (This spectrum is inside our `SolarSpectrum` class.) It also takes in the angle between the sun and the top of the sky dome (the *zenith angle*), your altitude, and the *atmospheric turbidity*, which is essentially a measure of how polluted the air is. A reasonable value for turbidity is around 2.2; going lower than 1.8 or higher than 20.0 will cause the math to start breaking down. This class will reward you with spectra of the *direct irradiance* of the sunlight transmitted through the atmosphere (this will become the diffuse component of your light source for the sun) and the *scattered irradiance* (which will become the ambient component of your light source).

The meat of the simulation is in `Spectrum::ApplyAtmosphericTransmittance`. This method iterates over samples of the visible spectrum from 380 nm to 720 nm. For each wavelength, we simulate the effects of several components of the atmosphere on how that wavelength is transmitted and scattered by the atmosphere. We can multiply together the transmittances from each component to arrive at a final transmittance for the given wavelength, and multiply that by the sun's irradiance at that wavelength. The scattered components are added together and then multiplied by the sun's irradiance. Once we're done with all of the wavelengths of the visible spectrum, we can convert this spectrum into RGB values for direct and ambient natural light.

It starts by computing the *air mass* for the given solar angle, which represents how much atmosphere the sunlight needs to pass through before it gets to the camera. The air mass M for a given solar zenith angle Z (in degrees) is given by

$$M = \frac{1.0}{\cos Z + 0.50572(93.885 - Z)^{-1.6364}}$$

The lower the sun is, the more air will scatter its sunlight. We multiply the air mass by the *isothermal effect*, which is a fancy way of saying that the higher you are, the less atmosphere there is. As your altitude increases, lower air masses will result in less light being scattered and more direct light reaching you; modeling this may yield effects such as the sky darkening as you start to enter space. The isothermal effect is given by $e^{-a/H}$, where a is the altitude above sea level, and H is the "pressure scale height" of 8435 meters.

Then, various components of the atmosphere are treated independently and their effects combined together at the end. The main component is Rayleigh scattering which is caused by the molecules of air itself; it is the reason the sky is blue, and your scattered light will be a bit blue as a result. The light transmitted by Rayleigh scattering T_R for a given wavelength λ in micrometers is given by

$$T_R = e^{\frac{-M}{\lambda^{4(1.15.6406-1.335/\lambda^2)}}}$$

Next comes the effect of *aerosols*, or larger particulate matter—this is affected by the turbidity T you passed in; more aerosols mean redder sunsets. The light transmitted by aerosols T_A is given by

$$\beta = 0.4608T - 0.04586$$

$$C_1 = \beta(2\lambda^{-\alpha})$$

$$T_A = e^{-C_1 M}.$$

The value of α may be set to 1.140 for rural environments. For better accuracy, it's really 1.0274 for wavelengths less than 500 nm, and 1.2060 otherwise.

We have computed the amount of light transmitted by Rayleigh and aerosol effects for each wavelength, which will give us our direct sunlight. For ambient sunlight, we also need to compute the scattered light. Scattered sunlight is a little trickier. First, we

need to compute an aerosol scattering transmission term T_{As} , an aerosol absorption transmission term T_{Aa} , the log of an aerosol asymmetry factor A , a constant F_S and a constant C_2 :

$$\Omega = 0.945e^{-0.095 \log(\lambda/4)^2}$$

$$T_{As} = e^{-\Omega C_1 M}$$

$$T_{Aa} = e^{(\Omega - 1.0) C_1 M}$$

$$C_2 = T_{Aa} \cos Z$$

$$A = \log 0.35$$

$$A_{FS} = 1.459A + 0.1595A^2 + 0.4129A^3$$

$$B_{FS} = 0.0783A - 0.3824A^2 - 0.5874A^3$$

$$F_S = 1 - \frac{1}{2} e^{-A_{FS} \cos Z - B_{FS} \cos^2 Z}.$$

The scattered light term ray D_{ray} for Rayleigh scattering is then given by

$$D_{\text{ray}} = \frac{C_2(1 - T_R^{0.95})}{2}.$$

The scattered term D_{aer} for aerosols is

$$D_{\text{aer}} = C_2 T_R^{1.5} (1 - T_{As}) F_S.$$

The total scattered irradiance can then be derived by adding D_{ray} and D_{aer} multiplying it by the sun's irradiance for the given wavelength. Modeling Rayleigh and aerosol effects will be accurate enough for most applications. The sample code also models the effects of water vapor, ozone, mixed gas, reflection from the ground, and some effects specific to wavelengths under 450 nm, but these effects are all small during daytime. However, simulating ozone scattering will improve the realism of your sunsets.

An important limitation of these equations is that they break down at zenith angles

over 90 degrees—that is, as soon as the sun drops below the horizon. Civil twilight is defined as the point where the sun is 6 degrees below the horizon, so to avoid a discontinuity at 90 degrees, you'll want to interpolate the direct and scattered sunlight between its value at 90 degrees and 0 at 96 degrees. Light doesn't really fall off linearly during this time; better implementations would use a lookup table based on experimental data of twilight luminance for given solar angles below the horizon. The nautical almanac [5] listed in the references is one source of this data.

The resulting spectra for transmitted and scattered sunlight are then converted to a CIE XYZ color. You will need to know a little about color theory to know what's going on here, but the important thing is that XYZ contains chromaticity information as well as luminance information. As such, it may represent "high dynamic range" colors, and the sun certainly qualifies as a high lighting value. We will map this light down to something displayable on a monitor, but first we also need to add in the light from the moon.

15.3 Moonlight and Other Nighttime Light Sources

Moonlight gets scattered through the atmosphere in exactly the same way as sunlight; the only difference is that instead of starting with a fixed spectrum of the extraterrestrial solar light source, we must generate the moon's spectrum algorithmically, since it varies depending on its phase and its distance from the Earth.

Fortunately, our `Ephemeris` class will give us that information. Moonlight consists of two components: light reflected from the Earth off the moon ("Earthshine") and light reflected from the sun off the moon. Both depend on the phase of the moon, as expressed by its phase angle Φ . Earthshine is given by

$$E_{em} = 0.095 \left[1 - \sin \frac{\pi - \varphi}{2} \tan \frac{\pi - \varphi}{2} \ln \left(\cot \frac{\pi - \varphi}{4} \right) \right].$$

This value becomes a component of the expression for computing the total moonshine irradiance seen from the Earth:

$$E_m = \frac{2Cr_m^2}{3d^2} \left\{ E_{em} + E_{sm} \left[1 - \sin \frac{\varphi}{2} \tan \frac{\varphi}{2} \log \left(\cot \frac{\varphi}{4} \right) \right] \right\}.$$

Here, d is the distance from the Earth to the moon returned from our `Ephemeris` class, r_m is the radius of the moon (1738.1×10^3 m), E_{sm} is the irradiance of the sun at the moon (1905 W/m²), and C is the average albedo of the moon (0.072).

To turn this into a spectrum that you can pass through the Bird spectral model, first convert W/m² from the equation above to cd/m² using the approximate conversion factor of $683.0/3.14$. Then, linearly scale this value from 0.7 at the low end of the visible spectrum to 1.3 at the high end, normalizing the results to ensure the resulting spectrum adds up to the lunar irradiance you computed above. From there, you can treat moonlight just like sunlight, and model the moon as a second light source in the same manner as the sun.

Even when there is no moon out at night, there are sources of ambient illumination. Light from bright planets, zodiacal light, starlight, airglow, galactic light, and cosmic light can all be modeled, but are negligible ($\sim 2 \times 10^{-6}$ W/m²) compared to artificial light pollution in all but the most remote areas. To preserve some visibility on moonless nights, you will want to add an arbitrary light pollution term to your ambient illumination.

15.4 Tone-Mapping the Light

Now that you have the direct and scattered irradiance from the sun and moon expressed as XYZ colors, the challenge is to map these down to RGB values for lighting your scene.

The difference in luminance between a moonless night and high noon in the summer is more than ten orders of magnitude and cannot be directly displayed by any display device. Tone-mapping is required to capture the fact that your eyes adapt to the ambient light, allowing you to see on a moonlit night while not being blinded during the day. During the day, the cones in your eye create what is known as *photopic* vision. At night, your rods are responsible for *scotopic* vision. At dawn and dusk, both may be active to provide *mesopic* vision. Perceptual tone mapping works differently in each case. For example, things appear to look a little blue at night, which is an effect we can capture. Fortunately, this is a solved problem. Frédo Durand and Julie Dorsey at MIT presented a simple tone-mapping operator for this purpose in 2000, which we'll summarize here.

Perceptual tone-mapping requires knowledge of both the average luminosity of the scene (this is the *adaptation luminosity* that your eyes are adapted to), and the maximum luminance of the display device. The adaptation luminosity may be approximated by the Y component of the sum of the sun and moon's scattered light. The display's luminosity is generally set to 100 cd/m².

The Durand operator treats tone mapping independently for rods and cones; we'll use the same notation used in Durand's paper [3]. The adaptation luminosity for cones L_{waC} is simply the Y component of the scene's scattered light, as mentioned above. The display threshold for rods L_{waR} may be approximated by

$$L_{waR} = -0.702X + 1.039Y + 0.433Z,$$

where X, Y, and Z are the XYZ components of the scene's scattered light, which is the sum of the scattered light from the sun and the moon. The display luminosities L_{daC} and L_{daR} are set to 100 cd/m², but may be adjusted for brighter or darker scenes. The rod and cone thresholds from the scene are then mapped to rod and cone thresholds for the display, using a technique called *threshold mapping*. For both rods and cones, photopic, mesopic, and scotopic conditions are treated separately; the rod threshold is given by

ε_R and the cone threshold by ε_C :

$$\log \varepsilon_R(L_{aR}) \begin{cases} -2.86, & \text{if } \log L_{aR} \leq -3.94; \\ \log L_{aR} - 0.395, & \text{if } \log L_{aR} \geq -1.44; \\ (0.405 \log L_{aR} + 1.6)^{2.18} - 2.86, & \text{otherwise.} \end{cases}$$

$$\log \varepsilon_C(L_{aC}) \begin{cases} -0.72, & \text{if } \log L_{aR} \leq -2.6; \\ \log L_{aC} - 1.255, & \text{if } \log L_{aR} \geq 1.9; \\ (0.249 \log L_{aC} + 0.65)^{2.7} - 0.72, & \text{otherwise.} \end{cases}$$

What we really need is to compute two scaling values, one for rods (m_R) and one for cones (m_C):

$$m_R = \frac{\varepsilon_C(L_{daC})}{\varepsilon_R(L_{waR})}$$

$$m_C = \frac{\varepsilon_C(L_{daC})}{\varepsilon_C(L_{waR})}$$

We also need a scaling value k which is used to interpolate between full color perception from cones and blue-shifted monochromatic perception from rods in mesopic conditions. With the value σ set to 100 cd/m²,

$$k = \frac{\frac{3}{4}\sigma L_{waR}}{\sigma + L_{waR}}$$

Finally, we have everything we need to map the raw XYZ values of the direct and scattered light to something displayable. For the raw lighting value \mathbf{L} , the tone-mapped value lighting value \mathbf{L}' is given by

$$\mathbf{R} = -0.702\mathbf{L}_x + 1.039\mathbf{L}_y + 0.433\mathbf{L}_z$$

$$\mathbf{S} = (0.3, 0.3, 0.4)\mathbf{R}$$

$$\mathbf{L}' = \frac{\mathbf{L}(1 - k)m_C + \mathbf{S}(km_R)}{L_{daC}}$$

The intermediate value **S** above represents the scotopic color from the rods (with a perceptual blue-shift), which is blended with the full color **L** from the cones.

The final step is to convert the tone-mapped XYZ value into an RGB value. You may perform this conversion with many matrices found online that assume different white points. Here is one with good results (this is the HDTV rec. 709 matrix):

$$[R \ G \ B] = [X \ Y \ Z] \begin{bmatrix} 3.240479 & -0.969256 & 0.055648 \\ -1.53715 & 1.875991 & -0.204043 \\ -0.49853 & 0.041556 & 1.057311 \end{bmatrix}$$

Unless you're working in HDR space, you'll want to clamp the results to [0,1] for each color component.

The algorithms above are implemented in the class `LuminanceMapper` in the included sample code on the CD.

15.5 Implementation Notes

Although the computations described above can execute pretty quickly, there is no need to compute them every frame. The resulting light values should be cached until the time of day or location changes, and may even be precomputed for the range of zenith angles for a given location at load time.

With physically-based light sources, it becomes important that the materials on your scene's objects are also accurate. If the materials on your objects are set to 100% brightness for diffuse or ambient light, they will look too bright when using these techniques. Real-world materials do not reflect 100% of the incoming light, unless they are perfect mirrors. Work with your art staff to ensure your materials are reasonable.

These algorithms all assume a clear sky at the camera's location; for cloudy conditions, you'll need to attenuate the results. Since the amount of attenuation depends

on the thickness of the clouds, how much is really up to you. For really thick clouds, you may want to reduce the amount of direct light and increase the ambient, since the clouds will scatter the sunlight further.

Strictly speaking, scattered sunlight and moonlight is not really ambient light—it is more accurately modeled as directional light radiating perpendicular to the surface of the sky dome. Without some sort of global illumination scheme, however, your objects will likely appear too dark if the scattered light does not reflect between objects on the ground, and treating it as an ambient term will yield better results. There is certainly nothing to stop you from doing something more sophisticated with the resulting direct and scattered light that these techniques produce.

References

- [1] R. E. Bird and C. Riordan. "Simple Solar Spectral Model for Direct and Diffuse Irradiance on Horizontal and Tilted Planes at the Earth's Surface for Cloudless Atmospheres". Technical Report No. SERI/TR-215-2436, Solar Energy Research Institute, 1984.
- [2] Peter Duffett-Smith. *Practical Astronomy with your Calculator*. Cambridge University Press, 1988.
- [3] Frédo Durand and Julie Dorsey. "Interactive Tone Mapping". *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pp. 219–230.
- [4] Henrik Wann Jensen, Frédo Durand, Julie Dorsey, Michael M. Stark, Peter Shirley, and Simon Premože. "A Physically-Based Night Sky Model". *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, 2001, pp. 399–408.
- [5] Nautical Almanac Offices of the United Kingdom and the United States of America. *Explanatory Supplement to the Astronomical Ephemeris and the American Ephemeris and Nautical Almanac*. Her Majesty's Stationary Office, 1961.

16

Rendering Physically-Based Skyboxes

Frank Kane

Sundog Software, LLC

Overview

Simple, GPU-friendly algorithms exist for rendering realistic skyboxes in real time for any given time and location. This gem reviews the Preetham et al. model [3] for accurately distributing luminance throughout a sky, with some extensions to ensure natural-looking results. When paired with the gem "Physically Based Outdoor Scene Lighting", outdoor scenes with lighting that perfectly matches the sky become possible. Algorithms for procedurally generating realistic skyboxes are surprisingly simple, and enable continuous time of day effects in your engine.

16.1 Generating and Drawing the Skybox

First, some skybox basics: a skybox is just that—a cube that is always rendered around the camera's location such that it moves and rotates with the camera, which is colored to look like a sky. You may properly position it by simply zeroing out the translation components of your view matrix before drawing it.

Intuition would tell us to render the skybox as the first thing in your frame, as it's

infinitely distant and everything else will be drawn in front of it. You could just disable depth buffer reads and writes, draw the skybox instead of clearing the depth buffer, and go about rendering your scene. This simple approach, however, is not the most optimal, since you'll end up spending time filling your color buffer with a bunch of sky that will end up being overdrawn by your scene. In reality, you won't save anything on modern hardware by not clearing the depth buffer along with your color buffer at the beginning of the frame—in fact, it may hurt performance. A better approach is to keep clearing your color and depth buffer together, then draw your skybox as the last thing in your frame, with depth buffer reads enabled to prevent drawing parts of the sky that are not visible. If you use an infinite projection matrix while rendering the skybox, drawing the skybox last instead of first will render correctly and can gain you some performance.

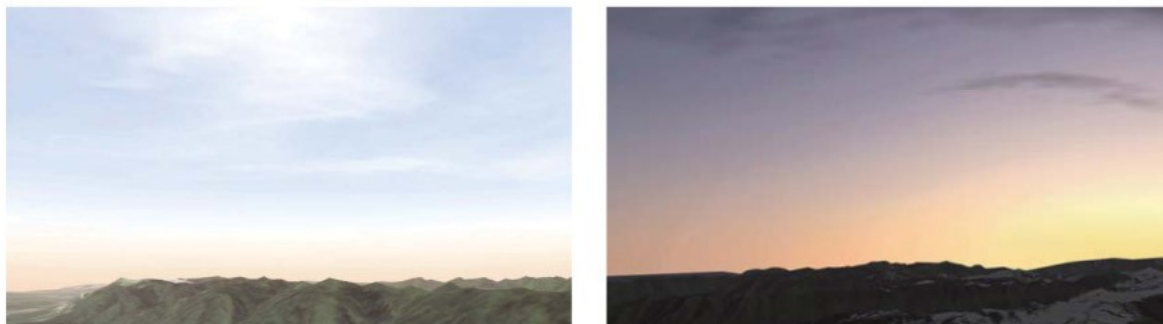


Figure 16.1: (See also Color Plates.) Physically-based skybox generated for (left) late morning and (right) twilight. (Images courtesy of Sundog Software, LLC.)

For procedurally generated skyboxes, we'll use a vertex program to render the sky just using vertex colors (disable texturing and enable smooth shading). This implies that the geometry of the skybox warrants some effort; you need to have enough vertices in the skybox to achieve convincing results, but too many may impact performance. In the images shown in Figure 16.1, each face of the skybox cube consists of a 40×40 grid. You may choose to have a higher vertex density near the horizon, as this is where sky colors

change more quickly; you may also choose to have higher vertex density vertically rather than horizontally, as the sky color changes less rapidly as you look around the horizon than as you look up toward the sky's zenith.

If you're certain that the bottom of your skybox will never be visible, you may omit this face—but as an engine developer, you shouldn't make this assumption. Your users might end up rendering the Earth from space using your skybox for a star-field and wonder why nothing is drawn underneath it. You may also want to allow your users to shift the skybox down by some amount instead of drawing it perfectly centered with the camera position; if the terrain in the scene doesn't actually extend all the way to the horizon, this is a simple way to cover up that fact.

The techniques in this gem will work just as well with a sky dome. However, a box is simpler, allows us to easily cull faces of the box that are not visible, and since we're using vertex colors instead of texture mapping, the corners of the box are not at all perceptible.

Using a conventional projection matrix, the size of the skybox must be chosen with care; you need to ensure that it falls within the near and far clipping planes of your view frustum. This becomes especially problematic with engines that dynamically adjust the near and far clip planes based on the scene's bounding volumes to maximize depth buffer resolution. An elegant way around this problem is to render the skybox using an infinite projection matrix and with w -coordinates of zero, assuring the far clip plane is irrelevant and the skybox is drawn at maximum depth. Care must be taken with this technique to avoid round-off error artifacts [1].

16.2 Computing the Skybox Vertex Colors

Back in 1993, Perez et al. developed a simple model for distributing luminance

across the sky, by fitting equations to experimental data [2]. In 1999, Preetham et al. extended this method to work in full color in a paper presented at Siggraph. The only input required is the position of the sun (or moon) in the sky, and the *turbidity* of the atmosphere. Turbidity is a measure of the particulate matter in the atmosphere; a reasonable value for realistic-looking scenes is around 2.2. Higher values imply more polluted skies, and will result in more dramatic sunsets.

The position of the sun or moon is specified in polar coordinates. θ_s is the angle from the up axis to the light source, and φ_s is the angle from the South axis (positive is toward East.) The `Ephemeris` class introduced in "Physically-Based Outdoor Scene Lighting" is useful for computing the position of the sun and moon for a given time and location.

The Perez model for distributing luminance across the sky is given by

$$F(\theta, \gamma) = (1 + Ae^{B/\cos\theta})(1 + C^{D\gamma} + E\cos^2\gamma).$$

We refer to this equation as the "Perez function". Here, θ is the angle of the skybox vertex from the up axis, and γ is the angle between the vertex and the sun or moon—your vertex program will compute these angles given the position of each skybox vertex. We'll discuss the constants A through E shortly.

To arrive at an actual luminance value Y_p for a given vertex, you need to compute the above function once for the vertex's position, and again for the position of the sun or moon (this only needs to be computed when the time of day changes, then cached):

$$Y_p = Y_z \frac{F(\theta, \gamma)}{F(0, \theta_s)}.$$

You'll also need the zenith luminance Y_z —although Preetham et al. provides a function for computing this in their paper, we've gotten better results by using the Y component of the tone-mapped scattered sunlight or moonlight derived in the gem "Physically-Based Outdoor Scene Lighting". Divide that luminance by 1000, since the

constants we're using here are in units of kcd/m^2 .

Computing the luminance Y_p for the skybox vertex is nice, but you want a color, not just a luminance. The trick is to work in xyY color space—this is a means of representing colors where the values x and y represent the color's chromaticity, and Y represents its luminance. For each vertex, you compute the function above three times—once for x , once for y , and once for Y , and then convert the resulting xyY color to RGB.

This means you'll need an xyY value for the zenith Y_z . If you have the zenith color in XYZ form, you'll need to convert XYZ to xyY using the relationship

$$x = \frac{X}{X + Y + Z}$$

$$y = \frac{Y}{X + Y + Z}.$$

The constants A , B , C , D , and E in the Perez function are themselves functions of the turbidity value T , and are different depending on whether you're computing the Perez function for x , y , or Y . These constants are given by

$$\begin{bmatrix} A_Y \\ B_Y \\ C_Y \\ D_Y \\ E_Y \end{bmatrix} = \begin{bmatrix} 0.1787 \\ -0.3554 \\ -0.0227 \\ 0.1206 \\ -0.0670 \end{bmatrix} T + \begin{bmatrix} -1.4630 \\ 0.4275 \\ 5.3251 \\ -2.5771 \\ 0.3703 \end{bmatrix}$$

$$\begin{bmatrix} A_x \\ B_x \\ C_x \\ D_x \\ E_x \end{bmatrix} = \begin{bmatrix} -0.0193 \\ -0.0665 \\ -0.0004 \\ -0.0641 \\ -0.0033 \end{bmatrix} T + \begin{bmatrix} -0.2592 \\ 0.0008 \\ 0.2125 \\ -0.8989 \\ 0.0452 \end{bmatrix}$$

$$\begin{bmatrix} A_y \\ B_y \\ C_y \\ D_y \\ E_y \end{bmatrix} = \begin{bmatrix} -0.0167 \\ -0.0950 \\ -0.0079 \\ 0.0441 \\ -0.0109 \end{bmatrix} T + \begin{bmatrix} -0.2608 \\ 0.0092 \\ 0.2102 \\ -1.6537 \\ 0.0529 \end{bmatrix}$$

These values only need to be recomputed if the simulated turbidity level changes, and should be passed into your vertex program as uniform parameters. The result of the Perez function for the sun position and the zenith xyY color should also be uniform parameters, as well as the position of the sun (or moon).

With these values and the position of each vertex in your skybox, you have everything you need for your physically-based skybox vertex program to compute xyY values for each sky vertex. The last step is to map this result to RGB values. To do this, first convert the xyY value to XYZ:

$$X = x \frac{Y}{y}$$

$$Z = (1 - x - y) \frac{Y}{y}.$$

If the zenith color you're using is already tone mapped, then further tone mapping is optional. The same tone mapping technique described in "Physically-Based Outdoor Scene Lighting" may be applied at this point, however, and will result in effects such as a warm glow surrounding a full moon in the sky.

Then, convert XYZ to RGB using your favorite conversion matrix. We use

$$[R \ G \ B] = [X \ Y \ Z] \begin{bmatrix} 3.240479 & -0.969256 & 0.055648 \\ -1.53715 & 1.875991 & -0.204043 \\ -0.49853 & 0.041556 & 1.057311 \end{bmatrix}$$

Other matrices tend to make the sky look a little green, which looks quite unnatural. If you're not working in HDR space, you'll want to scale down the resulting RGB values

to fit within $[0,1]$ if necessary.

The final tweak to the RGB value is important and often overlooked: gamma-correcting the color for the display. Your sky will appear too dark otherwise. Simply raise the final RGB color to the power of $1/\gamma$. A gamma value of 1.8 provides satisfying results.

A simple vertex program written in Cg that implements much of this algorithm is available with the book's sample code. Potential extensions of this program would include handling two light sources simultaneously (the sun and the moon) instead of a single dominant light source, incorporating volumetric fog effects into the sky, simulating overcast conditions, and the tone-mapping described above.

16.3 Integrating the Skybox with Your Scene

One challenge a physically-based skybox presents is that it becomes difficult to blend your distant scenery with the skybox, since the sky's colors may vary about the horizon. The usual trick of fogging distant terrain to blend into the sky may produce a visible seam, since the sky is not a constant color.

Preetham et al. describe an involved means for implementing physically-based atmospheric perspective effects on terrain that will match the skybox perfectly, but from a real-time performance standpoint this is impractical for dynamic scenes with complex geometry.

One solution is to blend the skybox itself to a fixed fog color at and below the horizon. If this fog color is chosen wisely, the sky/terrain boundary will be seamless, but you will lose some of the more dramatic effects at dawn and dusk with this approach.

Another solution is to sample the skybox color at the horizon directly in front of the camera every frame, and set your terrain's fog color to match it. The result won't be

perfectly seamless, but this is a good compromise for reasonable fields of view.

16.4 Embellishing Your Skybox

After drawing the skybox itself, you'll want to draw billboards representing the sun and the moon (preferably in the correct phase) in their proper locations. Both the sun and moon, by an amazing coincidence, cover a half a degree of the sky. However, if you render them at their physically accurate size, they'll seem much too small due to psycho-perceptual issues and typical fields of view that are unrealistically large—go ahead and draw them at whatever size looks right to you. Users also expect to see a large glare effect surrounding the sun, which will also help it to look bigger.

Another dramatic effect is rendering the stars and planets as part of your skybox. If you draw them as points with an additive blending mode, they'll start to emerge at dusk. Using the matrices in our `Ephemeris` class, your stars will move across the sky in a realistic manner. Data on the positions, magnitudes, and colors of the visible stars are readily available, and integrating this data into your skybox is a fun project.

This gem will render clear skies for any time of day and location, but clear skies are not the norm. Adding some clouds to the scene adds an extra level of realism. Rendering properly lit 3D volumetric clouds is a challenging task, but even a single large quad high in the sky with a cirrus cloud texture on it will go a long way.

References

[1] E. Lengyel. "Projection Matrix Tricks". Game Developer's Conference 2007.
http://www.terathon.com/gdc07_lengyel.ppt

[2] R. Perez, R. Seals, and J. Michalsky. "An All-Weather Model for Sky Luminance Distribution". *Solar Energy*, Volume 50, Number 3 (March 1993), pp. 235–245.

[3] A. J. Preetham, Peter Shirley, and Brian Smits. "A Practical Analytic Model for Daylight", *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, 1999, pp. 91–100.

17

Motion Blur and the Velocity-Depth-Gradient Buffer

Eric Lengyel

Terathon Software

Highlights

Motion blur adds a significant amount of realism to a rendered scene since our eyes are accustomed to seeing it when we look at moving objects in the real world. There are several techniques for producing motion blur in computer graphics, and they vary widely in both rendering speed and image quality. Temporal supersampling, in which multiple frames are rendered and then combined to form one image, can produce very accurate results, but its extreme rendering expense makes it impractical for real-time applications like games. Techniques using an accumulation buffer of some kind to store previous frames to be combined with the current frame are fast, but produce terrible results in terms of image quality.

There is a class of motion blur techniques that are based on calculating perpixel velocities and using them to collect many samples from the color buffer along the direction of motion. These techniques generally produce good results, but many of them produce a particular artifact that manifests itself as a fuzzy halo around foreground objects when the pixels behind them are moving quickly. The technique described in [1] makes no attempt to eliminate or reduce the appearance of this artifact. The method presented in [2] eliminates the artifact, but also eliminates some cases of

correct motion blur, and it comes with some significant limitations.

There is but one method that is both fast and capable of producing high-quality images, and it involves the use of a velocity buffer in conjunction with a post-processing shader to render a directional blur for pixels belonging to moving objects. A basic implementation of this concept produces adequate results for some applications, but it also produces the fuzzy halo artifact. This gem discusses an improvement to this motion blur technique that eliminates halo artifacts without also affecting cases where motion blur would be correctly rendered, producing images of much higher quality than is possible with other techniques. The method described here was originally implemented in the C4 Engine [3], and that is the source of the images shown in this gem.

17.1 Technique Overview

The technique described in this gem requires that a dedicated four-channel velocity buffer be allocated by the rendering system. For each frame we render, we fill this velocity buffer with information about the two-dimensional screen-space velocity of pixels belonging to each object to which we want to apply the motion blur effect. This is typically done early in the rendering process for a particular frame, and it happens independently of any previously rendered frames.

When rendering to the velocity buffer, there are three sources of motion that we need to consider. First, we must take the motion of the camera into account, and this motion affects all objects in the scene. Second, we must consider the motion that individual objects have as a whole. An object may be moving through space or rotating, and this motion can be captured by considering the object's transformation matrix for both the current frame and the preceding frame. Third, it's possible that the vertices composing an object's triangle mesh are themselves in motion. This frequently occurs with skinned character models, soft bodies, and cloth. Examples of motion blur arising

from camera movement and object movement are shown in Figure 17.1. An example of motion blur due to vertex movement within a single mesh is shown in Figure 17.2.



Figure 17.1: (See also Color Plates.) In the left image, motion blur resulting only from camera movement is shown. Notice how the ground and trees closer to the camera are blurred much more than distant objects. In the right image, motion blur resulting from rigid objects moving in the scene is shown. Both translational and rotational motion are visible in this still image. *(Images courtesy of Terathon Software LLC.)*



Figure 17.2: Motion blur resulting from vertex movement on a skinned character model. *(Image courtesy of Terathon Software LLC.)*

At the end of the rendering process for a single frame, we apply to the entire screen a post-processing shader that generates the motion blur effect using the data stored in the velocity buffer. This shader can usually be combined with other post-processing effects such as glow, distortion, and color matrix application. The shader that generates the motion blur reads the screen-space velocity for each pixel from the velocity buffer and uses it to choose a set of sample points at which the color buffer is then read. These color samples are distributed along the direction of the velocity, and they are spread over a larger distance for higher velocities. After the color samples are collected, they are combined to generate the final color for each pixel.

It is not always the case that we want to use all of the color samples for a particular pixel. In particular, if a fast-moving object passes behind a slow-moving or stationary object with respect to the camera, then the motion blur applied to pixels belonging to the background object should not sample pixels belonging to the foreground object. To prevent this from occurring, we must be able to determine when pixels belong to the same object and when they don't, while the post-processing shader is collecting color samples. This can be accomplished by storing additional information about the depth and the slope of surfaces in the velocity buffer.

In the two remaining available channels of the velocity buffer, we store the depth z of each pixel in camera space, and we store the magnitude of the two-dimensional gradient of the depth. These values give us the ability to calculate the minimum depth z_{\min} that a sample location must have in order to be considered part of the same surface as the pixel being blurred. The formula is

$$(17.1) \quad z_{\min} = z - r|\nabla z|,$$

where r is the distance from the sample location to the pixel being blurred. Color samples lying closer to the camera than this minimum depth are discarded. Figure 17.3 demonstrates how this technique eliminates artifacts that appear if the depth and

gradient are not considered.

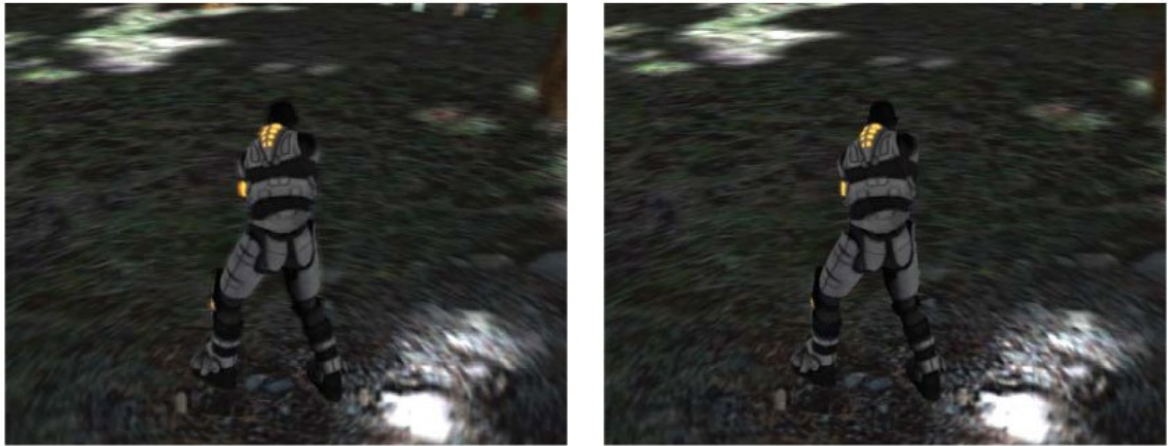


Figure 17.3: (See also Color Plates.) In these two images, the camera is rotating around the character, causing the ground to move across the screen while the character is almost completely still. In the left image, the depth and gradient information in the velocity buffer is not considered, and all color samples along the direction of the velocity are used. Notice the ghosting of the glowing parts of the character's armor and the fuzzy halo surrounding his legs. In the right image, the depth and gradient information in the velocity buffer is considered, and the rejection of the appropriate color samples eliminates the artifacts. (*Images courtesy of Terathon Software LLC.*)

17.2 Rendering to the Velocity-Depth-Gradient Buffer

At some point in time before post-processing is performed, we must fill a dedicated velocity-depth-gradient buffer with the information that will be used to generate the motion blur effect. Many modern engines render a depth-only pass at the beginning of a frame in order to maximize the effectiveness of hierarchical depth buffering capabilities built into the graphics hardware. This gives us a convenient place to also render our velocity information without having to pass vertex data through the rendering pipeline a second time. Since many engines also require a linear depth value

to be generated and stored early in the frame in order to render some types of special effects, it is doubly convenient that such a depth is one of the values we must calculate and store in the velocity buffer.

We render out velocity, depth, and gradient values into a floating-point buffer having 16 bits per channel. It is possible to implement the technique described in this gem using a conventional integer buffer with 8 bits per channel, but the small amount of available precision for the depth value in that case limits the practical range for which we can eliminate motion blur artifacts to an unacceptably short distance in front of the camera.

To calculate a two-dimensional screen-space velocity, we determine the screen-space positions for each vertex over two consecutive frames, subtract them, and then multiply by a normalizing scale factor. The homogeneous screen-space position $\mathbf{P}_{\text{screen}}$ of a vertex is given by

$$(17.2) \quad \mathbf{P}_{\text{screen}} = \mathbf{M}_{\text{viewport}} \mathbf{M}_{\text{project}} \mathbf{M}_{\text{camera}} \mathbf{M}_{\text{model}} \mathbf{P}_{\text{model}},$$

where $\mathbf{P}_{\text{model}}$ is the model-space vertex position, $\mathbf{M}_{\text{model}}$ is the matrix that transforms model-space points into world space, $\mathbf{M}_{\text{camera}}$ is the matrix that transforms worldspace points into camera space, $\mathbf{M}_{\text{project}}$ is the projection matrix for the camera, and $\mathbf{M}_{\text{viewport}}$ is the viewport transformation. The values of $\mathbf{M}_{\text{viewport}}$ and $\mathbf{M}_{\text{project}}$ are ordinarily constant from one frame to the next, but the values of $\mathbf{M}_{\text{camera}}$, $\mathbf{M}_{\text{model}}$, and $\mathbf{P}_{\text{model}}$ can change. Thus, it is necessary to store the $\mathbf{M}_{\text{camera}}$ matrix used by the camera during the preceding frame, and it is necessary to store the $\mathbf{M}_{\text{model}}$ matrix used by each model during the preceding frame. If the model-space vertex positions can change for a particular model (for example, on a skinned character), then the entire array of vertex positions used by that model during the preceding frame must also be stored.

When rendering an object into the velocity buffer, we calculate the product of the four matrices in Equation (17.2) to construct the matrix $\mathbf{M}_{\text{motion}}$ for both the preceding

frame and the current frame as follows:

$$(17.3) \quad \begin{aligned} \mathbf{M}_{\text{motion}}^A &= \mathbf{M}_{\text{viewport}} \mathbf{M}_{\text{project}} \mathbf{M}_{\text{camera}}^A \mathbf{M}_{\text{model}}^A, \\ \mathbf{M}_{\text{motion}}^B &= \mathbf{M}_{\text{viewport}} \mathbf{M}_{\text{project}} \mathbf{M}_{\text{camera}}^B \mathbf{M}_{\text{model}}^B. \end{aligned}$$

The superscript A indicates a matrix belonging to the preceding frame, and the superscript B indicates a matrix belonging to the current frame. The two products $\mathbf{M}_{\text{motion}}^A$ and $\mathbf{M}_{\text{motion}}^B$ are sent to the GPU as parameters that can be accessed by the vertex shader. As shown in Listing 17.1, these matrices are used in the vertex shader to calculate two homogeneous screen-space positions for each vertex and store them in texture coordinates that are interpolated as triangles are rasterized. The vertex shader shows the same position being transformed for both frames A and B, but in the case that $\mathbf{P}_{\text{model}}$ is not constant, a second vertex position array must be specified and used when calculating the position for frame A.

Listing 17.1: This GLSL vertex shader first transforms the vertex position for the current frame to homogeneous clip-space coordinates in the ordinary manner using the model-view-projection (MVP) matrix. The shader then transforms the vertex position into screen space for both the preceding frame using the matrix motionA and the current frame using the matrix motionB. The resulting positions are output as texture coordinates that will be read by the fragment shader.

```
uniform mat4    mvp, motionA, motionB;

void main()
{
    // Transform the position using the ordinary MVP matrix.
    gl_Position = mvp * gl_Vertex;

    // Transform the position into screen space using the motion matrix
```

```
// from the preceding frame (A) and the current frame (B).
gl_TexCoord[0] = motionA * gl_Vertex;
gl_TexCoord[1] = motionB * gl_Vertex;
}
```

The matrix multiplications performed by the vertex shader produce two four-dimensional homogeneous screen-space positions. It is important that these positions be interpolated in homogeneous form and that the perspective divide by the w -coordinate occurs in the fragment shader. Otherwise, the interpolated positions in the interiors of triangles would be incorrect, especially for triangles having vertices that lie behind the camera.

In the fragment shader used when rendering to the velocity buffer, we obtain two-dimensional screen-space positions for frames A and B by dividing the homogeneous interpolated positions $\mathbf{P}_{\text{screen}}^A$ and $\mathbf{P}_{\text{screen}}^B$ by their w -coordinates, as shown in Listing 17.2. Subtracting these positions then produces a screen-space velocity \mathbf{V} through the formula

$$(17.4) \quad \mathbf{V} = \frac{\mathbf{P}_{\text{screen}}^B}{(\mathbf{P}_{\text{screen}}^B)_w} - \frac{\mathbf{P}_{\text{screen}}^A}{(\mathbf{P}_{\text{screen}}^A)_w},$$

where only the x and y components of the velocity are calculated.

Listing 17.2: This GLSL fragment shader calculates the screen-space velocity and writes it to the red and green components of the output color. The `velocityScale` parameter holds the value of s/r_{max} shown in Equation (17.6). The depth is taken directly from the w -coordinate of the current position and is written to the blue component of the output color. The gradient of the depth is calculated using the hardware derivative functions, and the larger of its components is written to the alpha component of the output color.

```
uniform vec2    velocityScale;

void main()
{
    // Divide by the w-coordinates to get 3D positions.
    vec2 posA = gl_TexCoord[0].xy / gl_TexCoord[0].w;
    vec2 posB = gl_TexCoord[1].xy / gl_TexCoord[1].w;

    // Subtract the positions and scale to get velocity.
    vec2 veloc = (posB.xy - posA.xy) * velocity_scale;

    // Clamp the velocity to a max magnitude of 1.0.
    float vmax = max(abs(veloc.x), abs(veloc.y));
    gl_FragColor.xy = veloc / max(vmax, 1.0);

    // Pass the current depth through.
    gl_FragColor.z = gl_TexCoord[1].w;

    // Calculate the max component of the depth gradient.
    vec2 grad = vec2(ddx(gl_TexCoord[1].w), ddy(gl_TexCoord[1].w));
    gl_FragColor.w = max(abs(grad.x), abs(grad.y));
}
```

The magnitude of the velocity \mathbf{V} is unbounded, but we can only read a limited number of color samples per pixel in the post-processing phase, and we don't want them to be too far away from the pixel being processed. Thus, to ensure a smooth blur, we need to impose some kind of bounds on the velocity's size. We first divide the velocity by the maximum distance r_{\max} that we want to allow between a pixel's location and any color sample used to blur it. The value $1/r_{\max}$ is a constant that is passed to the fragment shader as a parameter by which the velocity is multiplied. We can also include in this parameter a normalization factor s that accounts for the time in between two frames

and adjusts the overall intensity of the motion blur effect. We define s as

$$(17.5) \quad s = \frac{t_0}{\Delta t} m ,$$

where t_0 is the target time interval between frames, Δt is the actual time between the preceding frame and the current frame, and m is an adjustable factor that controls the motion blur intensity. The scaled screen-space velocity \mathbf{V}' is given by

$$(17.6) \quad \mathbf{V}' = \frac{s}{r_{max}} \mathbf{V} .$$

After applying this scale factor, we clamp the velocity's magnitude to the range $[0,1]$ using the following formula to preserve its direction:

$$(17.7) \quad \mathbf{V}_{final} = \frac{\mathbf{V}'}{\max\{|V'_x|, |V'_y|, 1\}} .$$

The x and y components of the velocity \mathbf{V}_{final} are stored in the red and green channels of the color output to the velocity buffer.

What remains is to write the depth and gradient information to the blue and alpha channels of the velocity buffer. The linear camera-space depth is supplied by the w -coordinate of the position for the current frame, and it is simply copied to the blue channel of the output color. To obtain the gradient of the depth, we query the hardware for the derivatives of the position's w -coordinate in both the x and y screen directions. To achieve slightly higher performance, we output the larger absolute value of the two derivatives in the alpha channel instead of computing the actual magnitude of the gradient. (That is, we compute the maximum norm instead of the Euclidean norm.) In the fragment shader shown in Listing 17.2, the gradient magnitude g stored in the alpha channel is given by

$$(17.8) \quad g = \max \left\{ \left| \frac{\partial z}{\partial x} \right|, \left| \frac{\partial z}{\partial y} \right| \right\}.$$

17.3 Rendering the Post-Processing Effect

At the end of a frame, the motion blur effect is generated for the final image by rendering a post-processing shader over the entire screen. This shader uses the information in the velocity-depth-gradient buffer to select a set of sample locations from which the color buffer is read. All of the color samples that satisfy the minimum depth requirement are averaged together to produce the final color for each pixel. Since the sample locations are derived from the magnitude and direction of the velocity that a pixel possesses, the result is an image containing convincing motion blur.

The fragment shader shown in Listing 17.3 performs the motion blur operation. It starts by sampling the color buffer at the pixel location being rendered, which we call the "center pixel", and initializing the number of valid samples to one. The number of valid samples is stored in the *w* component of the color, and the sample at the center pixel is always valid. This particular implementation takes eight additional equally weighted samples from the color buffer at equally spaced intervals in the direction parallel to the velocity. There is some freedom in choosing the number of samples and their weights, and some implementations may even decide to take a variable number of color samples based on the magnitude of the velocity.

Listing 17.3: This GLSL fragment shader applies the motion blur effect in the post-processing pass. The nine color samples are accumulated in the *x*, *y*, and *z* components of the color vector, and the number of valid samples is stored in the *w* component of the color vector. The value of *minDepth* is calculated using Equation (17.9), and only samples having a depth at least this far from the camera plane are used to generate the final blurred pixel.

```
#extension GL_ARB_texture_rectangle : require

uniform sampler2DRect    colorTexture;
uniform sampler2DRect    velocityTexture;

void main()
{
    vec4    color, sample;

    // Read the center sample from the color buffer.
    color.xyz = texRECT(colorTexture, gl_FragCoord.xy).xyz;
    color.w = 1.0;

    // Read the velocity buffer at the current pixel.
    float4 velo = texRECT(velocityTexture, gl_FragCoord.xy);

    // Calculate the minimum depth for other color samples.
    float minDepth = velo.z - max(velo.w, 0.001) * 7.0;

    // Initialize constant sample weight.
    sample.w = 1.0;

    // Calculate coordinates for first sample on either side.
    vec4 coord = velo.xyxy * vec4(1.75, 1.75, -1.75, -1.75) +
        gl_FragCoord.xyxy;

    // Read a color and depth at the sample location.
    sample.xyz = texRECT(colorTexture, coord.xy).xyz;
    float depth = texRECT(velocityTexture, coord.xy).z;

    // Add the sample to the final color if it's depth is great enough.
    if (depth >= minDepth) color += sample;
```

```
// Grab the sample on the opposite side of the center pixel.
sample.xyz = texRECT(colorTexture, coord.zw).xyz;
depth = texRECT(velocityTexture, coord.zw).z;
if (depth >= minDepth) color += sample;

// Calculate coordinates for second pair of samples.
coord = velo.xyxy * vec4(3.5, 3.5, -3.5, -3.5) + gl_FragCoord.xyxy;
sample.xyz = texRECT(colorTexture, coord.xy).xyz;
depth = texRECT(velocityTexture, coord.xy).z;
if (depth >= minDepth) color += sample;

sample.xyz = texRECT(colorTexture, coord.zw).xyz;
depth = texRECT(velocityTexture, coord.zw).z;
if (depth >= minDepth) color += sample;

// Calculate coordinates for third pair of samples.
coord = velo.xyxy * vec4(5.25, 5.25, -5.25, -5.25) +
    gl_FragCoord.xyxy;
sample.xyz = texRECT(colorTexture, coord.xy).xyz;
depth = texRECT(velocityTexture, coord.xy).z;
if (depth >= minDepth) color += sample;

sample.xyz = texRECT(colorTexture, coord.zw).xyz;
depth = texRECT(velocityTexture, coord.zw).z;
if (depth >= minDepth) color += sample;

// Calculate coordinates for fourth pair of samples.
coord = velo.xyxy * vec4(7.0, 7.0, -7.0, -7.0) + gl_FragCoord.xyxy;
sample.xyz = texRECT(colorTexture, coord.xy).xyz;
depth = texRECT(velocityTexture, coord.xy).z;
if (depth >= minDepth) color += sample;

sample.xyz = texRECT(colorTexture, coord.zw).xyz;
```

```

depth = texRECT(velocityTexture, coord.zw).z;
if (depth >= minDepth) color += sample;

// Total weight of used color samples is in the w-coordinate.
// Divide by it to get the final averaged color.
gl_FragColor.xyz = color.xyz / color.w;
}

```

The velocity-depth-gradient buffer is read at the location of the center pixel, and the minimum depth required for all color samples is calculated as

$$(17.9) \quad z_{\min} = z - r_{\max} \max\{g, 0.001\},$$

where z is the depth stored in the blue channel of the buffer, g is the depth gradient given by Equation (17.8) stored in the alpha channel of the buffer, and r_{\max} is the largest distance between the center pixel and a sample location. This is a little different from Equation (17.1) because we use the maximum sample distance r_{\max} to compute a single minimum depth z_{\min} that is used for all sample values. We clamp the minimum value of the gradient to 0.001 so that precision errors don't prevent the motion blur effect from working on surfaces that are nearly perpendicular to the view direction.

The value of r_{\max} is 7.0 in the fragment shader shown in Listing 17.3, and color samples are taken at offsets given by the velocity vector multiplied by the values 1.75, 3.5, 5.25, and 7.0. At each sample location, the velocity buffer is read, but only to fetch the depth at that sample location and compare it to z_{\min} . (Velocity and gradient information is only used at the center pixel.) For any sample satisfying $z \geq z_{\min}$, we add the color sample to the final color and add one to the number of valid samples (in the w component of the color). After all samples have been taken, we divide the final color by the number of valid samples that have been accumulated and output the result.

17.4 Grid Optimization

The fragment shader presented in Listing 17.3 produces very precise results, but it can be unnecessarily expensive for large parts of the scene. When it is known that a region of the screen contains pixels that are all moving at similar speeds relative to the camera, a simpler shader that does not consider depth can be used in order to increase overall performance. The use of the full implementation can be limited to those areas of the screen in which slow-moving foreground objects are expected to appear.



Figure 17.4: (See also Color Plates.) In this image, the viewport is partitioned into a grid of 16×12 cells. The depth and gradient information is only used in the highlighted cells surrounding the character since that is where foreground objects are likely to be moving slowly relative to the background. (Image courtesy of Terathon Software LLC.)

In Figure 17.4, we have divided the screen into a 16×12 grid of rectangular cells.

Before rendering the post-processing shader, we determine whether any foreground objects might be slow moving relative to the background and mark cells covered by those objects as requiring the full-blown shader. This would typically be done when objects are being rendered into the velocity buffer near the beginning of the frame. In the post-processing phase, we apply the simpler, faster shader to cells that have not been marked.

References

- [1] Gilberto Rosado."Motion Blur as a Post-Processing Effect". *GPU Gems 3*, Addison-Wesley, 2008.
- [2] Ben Padget."Efficient Real-Time Motion Blur for Multiple Rigid Objects". *ShaderX7*. Charles River Media, 2009.
- [3] C4 Engine. <http://www.terathon.com/c4engine/>

18

Fast Screen-Space Ambient Occlusion and Indirect Lighting

László Szirmay-Kalos, Balázs Tóth, and Tamás Umenhoffer

Budapest University of Technology and Economics

18.1 Introduction

A physically correct approach to rendering would be the solution of the rendering equation, but it is too expensive computationally when dynamic scenes are processed in real time. Thus in practice, we prefer approximations that can be efficiently evaluated. A usual simplification is the local illumination model, which computes only the direct contribution of the light sources and adds a constant ambient term for the missing indirect illumination. However, constant ambient lighting ignores the geometry of the scene, which results in plain and unrealistic images. We need better compromises between the rendering equation and the classical ambient model.

Local approaches examine only a neighborhood of the shaded point during illumination calculation. The *obscurances method* [10, 2], which is also called the *ambient occlusion* [1, 6] computes just how "open" the scene is in the neighborhood of a point, and scales the ambient light accordingly. To compute occlusions in real time, the method called *screen-space ambient occlusion* [5] examines the height field defined

by the current content of the depth buffer instead of the real scene's geometry. Thus, when a point is shaded, the required geometric information about the rest of the scene is fetched from a depth texture. The classical ambient occlusion approach assumes that no illumination comes from nearby occluders. However, using the albedo or the color of these points, local indirect lighting can also be approximated [10, 8].

In this article, we present a simplified illumination model that is derived from the rendering equation. The model consists of two parts, the ambient occlusion part describing the influence of the distant part of the scene, and the indirect illumination part, which is responsible for local interactions. Both parts are directional integrals, which would need many discrete samples for an accurate estimation. In order for the efficient evaluation, we transform these integrals. The ambient occlusion integral is first transformed to a volumetric integral, which is first evaluated along the depth coordinate analytically, then the remaining integral over the disk perpendicular to the viewing direction is obtained numerically. The indirect lighting integral is expressed from the stable ambient occlusion estimate. Thus, our method is more general than [10] since it also takes into account the one-bounce of the direct lighting, and it is more robust than [8] since it does not include infinite variation form factors in the approximation.

The proposed method falls into the category of screen-space techniques since we assume that the depth buffer is the sampled representation of the scene geometry and the color buffer stores the radiance values of the represented surfaces. However, we work in camera space rather than in screen space to obtain correct distance and angle values.

The input of our rendering method includes the textures of camera-space depth values, normal vectors of the visible points, and the color buffer storing the radiances due to direct lighting. From these input textures, a deferred shading algorithm computes the radiances of the visible points, taking into account ambient occlusion and local indirect illumination.

18.2 A General Ambient Illumination Model

Let us assume that the surfaces are diffuse. According to the rendering equation, the *reflected radiance* L^r at a *shaded point* \mathbf{s} can be obtained as an integral of directions $\boldsymbol{\omega}$ running over the unit hemisphere Ω above the surface:

$$L^r(\mathbf{s}) = \int_{\Omega} L^{in}(\mathbf{s}, \boldsymbol{\omega}) \frac{a(\mathbf{s})}{\pi} (\mathbf{N}_s \cdot \boldsymbol{\omega}) d\boldsymbol{\omega} .$$

where a is the *albedo* of the diffuse surface, \mathbf{N}_s is the unit normal at the shaded point, and $L^{in}(\mathbf{s}, \boldsymbol{\omega})$ is the incident radiance of the shaded point from direction $\boldsymbol{\omega}$.

If an *occluder point* \mathbf{o} is visible from \mathbf{s} in the direction $\boldsymbol{\omega}$ (see Figure 18.1) and the space is not filled with participating media, then the incident radiance is equal to exiting radiance $L^r(\mathbf{o})$. If no surface is seen, then shaded point \mathbf{s} is said to be *open* in this direction, and the incident radiance is ambient intensity L^a . However, this does not meet our intuition and everyday experience that the effect of distant surfaces is replaced by their average. This experience is due to the fact that the actual space is not empty but is filled with a participating medium. If its extinction coefficient is σ and its albedo is 1, then the radiance along a ray of direction $\boldsymbol{\omega}$ changes according to the volumetric rendering equation

$$L^{in}(\mathbf{s}, \boldsymbol{\omega}) = e^{-\sigma D} L^r(\mathbf{o}) + (1 - e^{-\sigma D}) L^a ,$$

where D is the distance between the shaded and the occluder points. Note that in this equation factor $\mu(D)=1-e^{-\sigma D}$ and its complement $1-\mu(D)=e^{-\sigma D}$ express the effects of the ambient lighting and of the occluder on the shaded point, respectively. The effect of the occluder diminishes with the distance. The function μ is a *fuzzy measure* that defines how strongly direction $\boldsymbol{\omega}$ belongs to the set of open directions based on distance D of the occlusion at this direction.

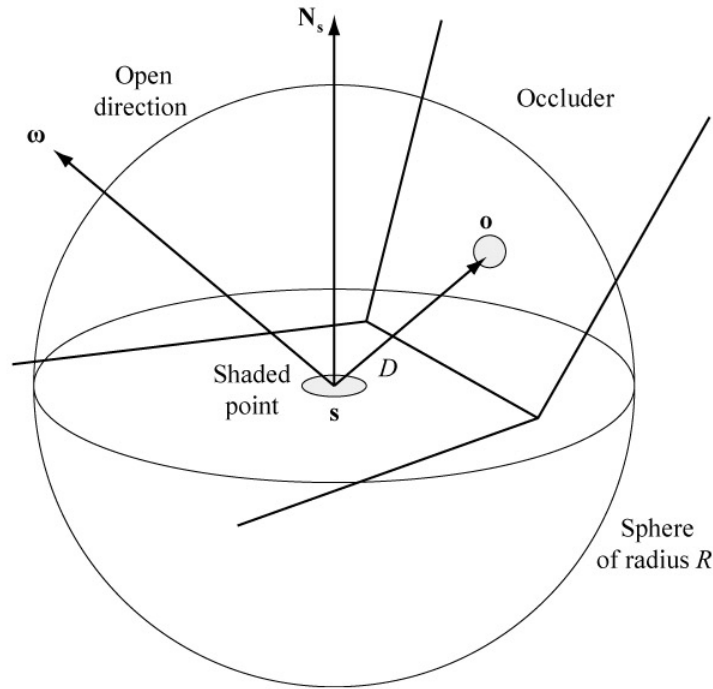


Figure 18.1: The shaded point s is the center of the neighborhood sphere. The radius of the sphere is R . Those directions ω where there is no intersection closer than R are called open. Point o is an intersection closer than R .

The exponential function derived from the physical analogy of participating media has a significant drawback [2]. As it is nonzero for arbitrarily large distances, very distant surfaces need to be considered that otherwise have a negligible effect. Thus, for practical fuzzy measures we use functions that are nonnegative, monotonically increasing from zero, and reach one at the finite distance R . This allows the consideration of only those occlusions that are nearby, i.e., in the *neighborhood sphere* of radius R . The particular value of R can be set by the application developer. When we increase this value, shadows due to ambient occlusions get larger and softer.

To define the fuzzy measure that increases from zero to one in $[0, R]$, we can use a

simple polynomial

$$\mu(D) = \left(\frac{D}{R}\right)^a.$$

Using this fuzzy measure, the reflected radiance of the shaded point can be expressed in the following way:

$$\begin{aligned} L^r(\mathbf{s}) &= a(\mathbf{s}) \left[\frac{L^a}{\pi} \int_{\Omega} \mu(D)(\mathbf{N}_s \cdot \boldsymbol{\omega}) d\boldsymbol{\omega} + \frac{1}{\pi} \int_{\Omega} (1 - \mu(D)) L^r(\mathbf{o})(\mathbf{N}_s \cdot \boldsymbol{\omega}) d\boldsymbol{\omega} \right] \\ &= a(\mathbf{s}) [L^a O(\mathbf{s}) + I(\mathbf{s})]. \end{aligned}$$

The first term of this expression is the ambient *occlusion* of the shaded point:

$$O(\mathbf{s}) = \frac{1}{\pi} \int_{\Omega} \mu(D(\boldsymbol{\omega})) (\mathbf{N}_s \cdot \boldsymbol{\omega}) d\boldsymbol{\omega}.$$

The second term is the irradiance due to nearby indirect lighting:

$$I(\mathbf{s}) = \frac{1}{\pi} \int_{\Omega} [1 - \mu(D(\boldsymbol{\omega}))] L^r(\mathbf{o})(\mathbf{N}_s \cdot \boldsymbol{\omega}) d\boldsymbol{\omega}.$$

This integral is traced back to the ambient occlusion. Replacing occluder radiance $L^r(\mathbf{o})$ by the average of surface radiance values in the neighborhood of the shaded point $\tilde{L}^r(\mathbf{s})$, we can express the irradiance as

$$I(\mathbf{s}) \approx \frac{1}{\pi} \int_{\Omega} [1 - \mu(D(\boldsymbol{\omega}))] \tilde{L}^r(\mathbf{s})(\mathbf{N}_s \cdot \boldsymbol{\omega}) d\boldsymbol{\omega} = \tilde{L}^r(\mathbf{s})(1 - O(\mathbf{s})).$$

18.3 Screen-Space Representation of the Scene

In global illumination computations, we need to know the geometry of other parts of the scene and the radiance values of points visible from the currently shaded point. However, GPUs are built according to the concept of local illumination and prefer shading each point independently of other parts of the scene. The only additional information that can be used is stored in textures.

Screen-space techniques assume that the height field defined by the current content of the depth buffer is an appropriate representation of the screen geometry, and the color buffer stores the radiance values of the represented points. Of course, the content of these buffers represents only the surfaces visible from the camera, but for local methods like ambient occlusion, this restriction is usually acceptable. In screen space, viewing rays are parallel to the z -axis, which greatly simplifies calculations. However, the transformation to this space is not angle and distance preserving (it is not even affine); thus, this space is not appropriate for angle and distance computation.

If we need to compute angles and distances, we should rather work in camera space, which means that we store camera-space z values and also the camera-space normal vectors of the visible points in textures. The transformation from world space to camera space is angle and distance preserving since it is basically a translation and a rotation; thus, these spaces are equivalent when distances and angles are computed. The disadvantage of camera space is that in the case of a perspective camera, the viewing rays are not parallel, but rather intersect each other at the origin of the coordinate system.

Thus, to solve the distortion problem of screen space and also keep the advantages of parallel rays, we work in camera space but use a quasi-orthogonal approximation. When large scale information is obtained, we follow the structure of camera space.

However, when smaller neighborhoods are explored, which happens during the evaluation of the ambient occlusion integral, we assume that in each small neighborhood, the viewing rays are parallel with the z -axis. This is an approximation, but is a reasonable compromise between accuracy and simplicity.

Indirect illumination computation requires those points that are visible from the shaded point, which can usually be obtained with ray tracing. Unfortunately, ray tracing is quite expensive computationally even for height fields, so we replace it by a simple test. As the shaded point belongs to the set of points that are visible from the camera, we require that the occluder point also be visible from the camera. Two points are visible from each other if the ray originating at one of the points has no intersection with any surfaces before it arrives in the other point. The requirement of being in the visible part of the height field, on the other hand, means that the ray intersects the surface zero, two, four, etc., times. If the neighborhood is small, and at most one intersection is possible, then the two criteria are similar.

18.4 Volumetric Ambient Occlusion

In order to find an efficient method for the evaluation of the ambient occlusion integral, we express it as a three-dimensional (i.e., volumetric) integral. First, the fuzzy measure is written as the integral of its derivative:

$$\mu(D) = \int_0^D \frac{d\mu(r)}{dr} dr$$

Substituting this integral into the ambient occlusion formula, we get

$$O(s) = \frac{1}{\pi} \int_{\Omega} \int_0^D \frac{d\mu(r)}{dr} (N_s \cdot \omega) dr d\omega .$$

Realizing that $r^2 dr d\omega = dV$ is a differential volume, the ambient occlusion can also be expressed as a volumetric integral

$$O(\mathbf{s}) = \frac{1}{\pi} \int_S \frac{d\mu(r)}{dr} \frac{1}{r^2} (\mathbf{N}_s \cdot \boldsymbol{\omega}) dV ,$$

where S contains those points of the solid hemisphere that are visible from the shaded point, and therefore also visible from the camera.

In our case, the occluder surface is a height field defined by the content of the depth buffer. Thus a point (x, y, z) belongs to the visible region if its z -coordinate is less than the value z^* stored in the depth buffer for the same (x, y) coordinates.

We compute the volumetric integral with differential elements having dz height and $dA = dx dy$ base area at the point (x, y) of the disk C of radius R and perpendicular to z -axis (see Figure 18.2) as

$$O(\mathbf{s}) = \frac{1}{\pi} \int_{x, y \in C} \int_{z_{\min}}^{z_{\max}} \frac{d\mu(r)}{dr} \frac{1}{r^2} (\mathbf{N}_s \cdot \boldsymbol{\omega}) dz dx dy = \frac{1}{\pi} \int_{x, y \in C} h(x, y, z_{\min}, z_{\max}) dx dy ,$$

where h is the integral over z given by

$$h(x, y, z_{\min}, z_{\max}) = \int_{z_{\min}}^{z_{\max}} \frac{d\mu(r)}{dr} \frac{1}{r^2} (\mathbf{N}_s \cdot \boldsymbol{\omega}) dz = \int_{z_{\min}}^{z_{\max}} \frac{d\mu(r)}{dr} \frac{1}{r^2} \frac{(\mathbf{o} - \mathbf{s}) \cdot \mathbf{N}_s}{r} dz .$$

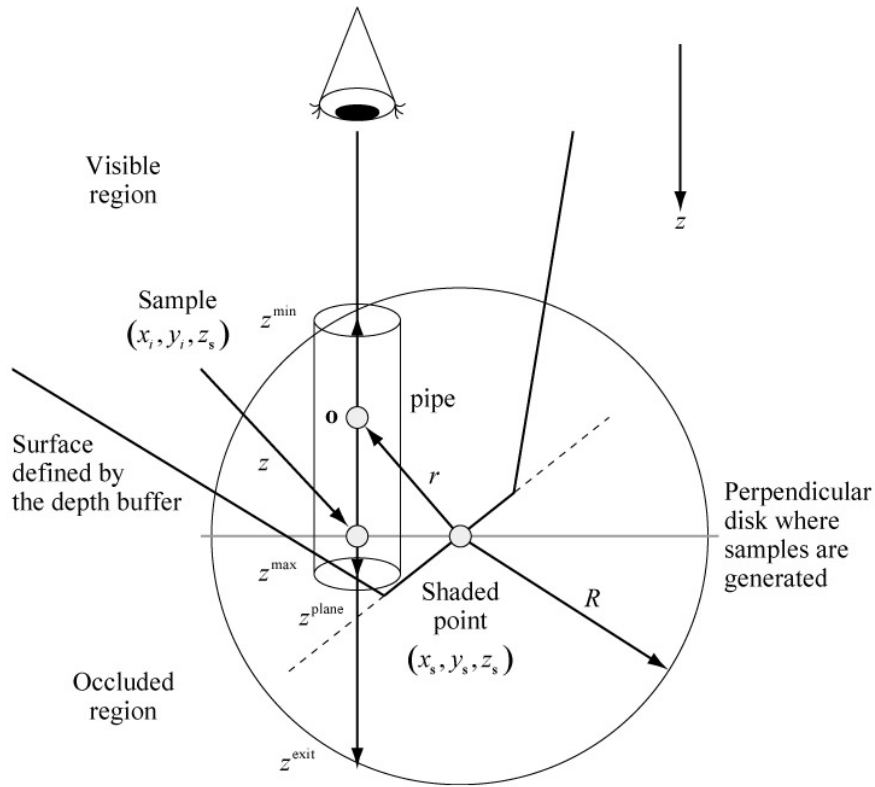


Figure 18.2: Evaluation of the volumetric integral in the visible part of the neighborhood sphere.

Recall that we are free to set the exponent of the fuzzy membership function. Classical ambient occlusion used a non-fuzzy measure, which corresponds to $\alpha = \infty$. Mendez [4] examined several exponents and concluded that $\alpha = 1/2$ is a good choice. Our criterion for setting the exponent will be the ease of the evaluation of the ambient occlusion integral. This integral can be evaluated analytically if we define the fuzzy membership function as $\mu(r) = (r/R)^\alpha$ with $\alpha = 4$. The center of the sphere is the shaded point whose coordinates are denoted by (x_s, y_s, z_s) . In this case,

$$\begin{aligned}
h(x, y, z_{min}, z_{max}) &= \frac{4}{R^4} \int_{z_{min}}^{z_{max}} (\mathbf{o} - \mathbf{s}) \cdot \mathbf{N}_s dz \\
&= \frac{4}{R^4} \left[z \left((x - x_s)N_s^x + (y - y_s)N_s^y \right) + \frac{(z - z_s)}{2} N_s^z \right]_{z_{min}}^{z_{max}}.
\end{aligned}$$

The integral over the disk is evaluated with numerical quadrature. The total volume of the visible part of the hemisphere is approximated by the sum of the volume of pipes. The axes of these pipes are parallel to the z -axis. The pipes are inside the hemisphere and may be limited by the height field of the depth buffer. To define the pipes, we sample n uniformly distributed points (x_i, y_i) in the disk of radius R . Thus, each pipe has the same cross section area $\Delta A = R^2 \pi / n$.

A line crossing the i -th sample point and being parallel with the z -axis enters the sphere at

$$z_i^{min} = z_s - \sqrt{R^2 - (x_i - x_s)^2 - (y_i - y_s)^2},$$

exits it at

$$z_i^{exit} = z_s + \sqrt{R^2 - (x_i - x_s)^2 - (y_i - y_s)^2},$$

and crosses the tangent plane of the shaded point at

$$z_i^{plane} = \frac{N_s^x(x_s - x_i) + N_s^y(y_s - y_i) + N_s^z z_s}{N_s^z},$$

The points on this line belong to the visible region when their z -coordinates are less than $z_i^{max} = \min(z_i^*, z_i^{exit}, z_i^{plane})$ and are greater than z_i^{min} . The contribution of the pipes to the volumetric integral of the ambient occlusion is

$$O(s) = \frac{1}{\pi} \int_{x,y \in C} h(x, y, z_{min}, z_{max}) dx dy \approx \frac{R^2}{n} \sum_{i=1}^n h(x_i, y_i, z_i^{min}, \min(z_i^*, z_i^{exit}, z_i^{plane})).$$

This quadrature is an approximation, and its error decreases if new sample points are added. However, computing the formula with many sample points reduces rendering speed. Thus, we consider two techniques, including weighted uniform sampling and interleaved sampling, that reduce the computation error without performance degradation.

Weighted uniform sampling [7] exploits the fact that if there is no occlusion in the neighborhood sphere, then ambient occlusion factor should be equal to one. If it is not, then the difference is due to the approximation error. So we compute not only the ambient occlusion from the samples but also the estimate of this factor assuming no occlusion at all. Ignoring occlusions, this factor is

$$1 \approx \frac{R^2}{n} \sum_{i=1}^n h(x_i, y_i, z_i^{min}, \min(z_i^*, z_i^{exit}, z_i^{plane})).$$

Dividing the formula for ambient occlusion by this approximation, we can compensate for the quadrature error; thus, a better estimate of ambient occlusion is

$$O(s) \approx \frac{\sum_{i=1}^n h(x_i, y_i, z_i^{min}, \min(z_i^*, z_i^{exit}, z_i^{plane}))}{\sum_{i=1}^n h(x_i, y_i, z_i^{min}, \min(z_i^{exit}, z_i^{plane}))}.$$

Interleaved sampling [3] takes advantage of the estimates in neighboring pixels. The method discussed so far has some error in each pixel, depending on the particular samples used. If we took different random numbers in neighboring pixels, dot noise would be present. Using the same random numbers in every pixel would make the error correlated and replace dot noise with stripes. Unfortunately, both stripes and pixel noise

are quite disturbing. Interleaved sampling uses different sets of samples in the pixels of a 4×4 pixel pattern. The errors in the pixels of a 4×4 pixel pattern are uncorrelated, which can be successfully reduced by a lowpass filter of the same size. When implementing the low-pass filter, we also check whether the depth difference between the current and neighboring pixels exceeds a given limit. If it does, we do not include the neighboring pixels in the averaging operation.

18.5 Indirect Lighting of the Near Geometry

The second part of the reflected radiance depends on the average radiance values of nearby surface points. As we calculate the ambient occlusion with random points in the neighborhood of the shaded point, the color of the frame buffer at these random points can be used to obtain the average reflected radiance. By inspecting the camera-space normal, we can also check whether the surface is oriented toward the shaded point, and ignore it in the average otherwise.

18.6 Implementation

The discussed algorithm is implemented as a fragment shader run in a deferred shading pass, as shown in Listing 18.1. For the sake of simplicity, we omitted parts related to interleaved sampling. The program receives the fragment position in texture space ($wPos$) and in 2D clipping space ($hPos$) as interpolants. By fetching from the texture map containing normal vectors and depth values (`depthMapSampler`) with the texture-space position, we obtain the camera-space normal vector and the depth value. The 2D clipping-space coordinates are also transformed back to camera space using the camera-space depth, which results in the shaded point s .

Listing 18.1: This fragment shader implements the algorithm discussed in this gem.

```
float4 psAO(float2 wPos : TEXCOORD0, float2 hPos : TEXCOORD1) : COLOR0
{
    wPos += pixelsize * 0.5; // Texture coordinates in [0,1]

    float3 N = tex2D(depthMapSampler, wPos).xyz; // Camera-space normal
    float depth = tex2D(depthMapSampler, wPos).a; // Camera-space depth

    // Compute camera-space position
    float4 pcamera = mul(float4(hPos, 0, 1), projMatrixInverse);
    pcamera.xyz /= pcamera.w;

    float3 s = pcamera.xyz * depth / pcamera.z; // Shaded point

    float O = 0; // Enumerator of ambient occlusion
    float Denom = 0; // Denominator of ambient occlusion
    float3 I = 0; // Irradiance

    for (int sampleidx = 0; sampleidx < sampleCount; sampleidx++)
    {
        float2 sample = AO_RAND[sampleidx].xy * R;
        float3 o = s + float3(sample.x, sample.y, 0); // Occluder

        pcamera = mul(float4(o, 1), projMatrix);

        float2 texCoord = pcamera.xy / pcamera.w;
        texCoord.y *= -1.0;
        texCoord = (texCoord + 1) / 2;
        float zstar = tex2D(depthMapSampler, texCoord).a;
        o.z = zstar; // Occluder's depth

        float d = sqrt(R * R - dot(sample.xy, sample.xy));
```

```

float zmin = s.z - d;
float zexit = s.z + d;
float zplane = s.z - dot(o.xy - s.xy, N.xy) / N.z;
zplane = max(zplane, zmin);
zexit = min(zplane, zexit);
float zmax = zexit;

Denom += (zmax - zmin) * (dot(o.xy - s.xy, N.xy) +
    (zmax + zmin - 2 * s.z) / 2 * N.z);

if (zstar < zmin - R) zstar = zexit;    // silhouette elimination
zmax = min(zexit, zstar);
zmax = max(zmax, zmin);

O += (zmax - zmin) * (dot(o.xy - s.xy, N.xy) +
    (zmax + zmin - 2 * s.z) / 2 * N.z);

if (zmax < zexit) // Occlusion happened?
{
    float3 No = tex2D(depthMapSampler, texCoord).xyz;
    if (dot(s - o, No) > 0)
        I += tex2D(colorMapSampler, texCoord).rgb;
}
}

O /= Denom;
I *= (1.0 - O) / sampleCount;
return float4(I, O);
}

```

Then the enumerator and denominator of the ambient occlusion formula and the irradiance are computed in variables `O`, `Denom`, and `I` in the loop executed `sampleCount` times. A single occluder sample `o` is generated from prepared 2D points uniformly

distributed in the unit disk ($AO_RAND[k]$). The depth map is fetched using the direction of the occluder, which results in occluder depth z_{star} . This depth is compared to the entry depth of the sphere z_{min} , exit depth z_{exit} , and that of the intersection with the tangent plane z_{plane} , determining the z_{min} - z_{max} interval where the function h is evaluated. In parallel, another integral is computed in $Denom$ that describes the unoccluded case. This integral will be used for error compensation. Note that we also check whether the occluder is much closer to the eye than the shaded point and ignore such occlusions, which would otherwise result in false silhouette edges. If near occlusion happens and, according to the occluder's surface orientation, it can illuminate the shaded point, then the occluder's color is inserted into the average color used for indirect illumination. This fragment shader returns with the average indirect illumination and the ambient occlusion of the fragment, which is then composited with the previously calculated direct illumination result.

As the compiler unrolls loops that contain `tex2D` calls, we may run out of registers when the sample number is high (it is greater than 12 in the specified hardware). If more samples are needed, the `tex2D` calls should be replaced by `tex2Dlod`, which does not force loop unrolling. Surprisingly, this replacement does not degrade the performance.

18.7 Results

The proposed methods have been implemented in DirectX/HLSL environment, and their performance has been measured on an Nvidia GeForce 8800 GTX GPU at 800×600 resolution. The rendering results of the harbor scene are shown in Figure 18.3. This scene can be rendered at 170 FPS if only directional light is computed. Taking 16 samples per pixel, the ambient occlusion and indirect lighting computation runs over 100 FPS. With 32 samples per pixel, the rendering speed drops to 60 FPS.

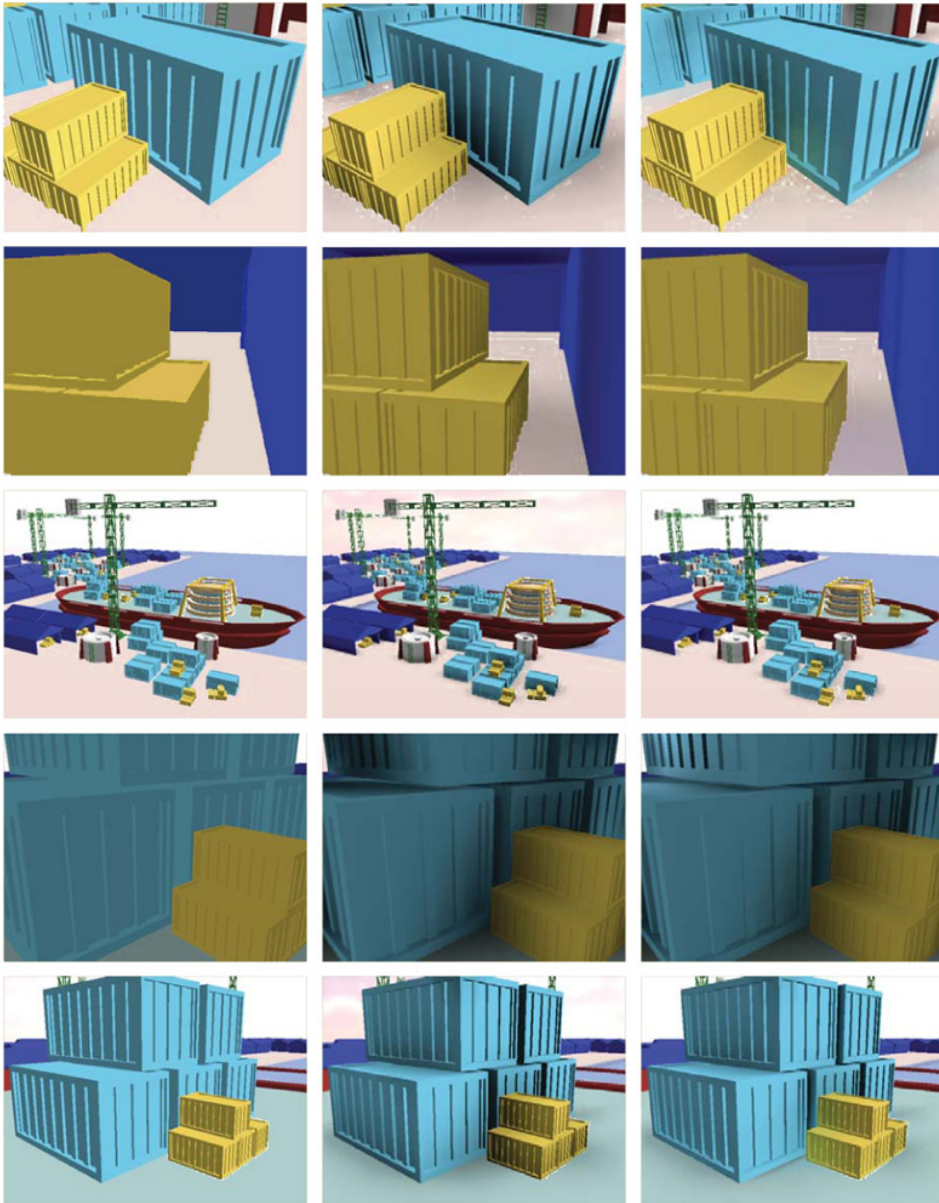


Figure 18.3: (See also Color Plates.) Rendering results of the harbor scene— (First column) direct lighting, (second column) direct lighting plus ambient occlusion, and (third column) direct lighting plus ambient occlusion and indirect lighting.

References

- [1] H. Landis. "Production-ready global illumination". SIGGRAPH Course notes 16, 2002. <http://www.debevec.org/HDRI2004/landis-S2002-course16-prodreadyGI.pdf>
- [2] A. Iones, A. Krupkin, M. Sbert, and S. Zhukov. "Fast realistic lighting for video games". *IEEE Computer Graphics and Applications*, Volume 23, Number 3 (May 2003), pp. 54–64.
- [3] A. Keller and W. Heidrich. "Interleaved sampling". *Rendering Techniques 2001 (Proceedings of the 12th Eurographics Workshop on Rendering)*, 2001, pp. 269–276.
- [4] A. Méndez, M. Sbert, and J. Catá. "Real-time obscurances with color bleeding". *SCCG '03: Proceedings of the 19th Spring Conference on Computer Graphics*, ACM, 2003, pp. 171–176.
- [5] M. Mittring. "Finding next gen — CryEngine 2". *Advanced Real-Time Rendering in 3D Graphics and Games course*, SIGGRAPH 2007, pp. 97–121.
- [6] M. Pharr and S. Green. "Ambient occlusion". *GPU Gems*, Addison-Wesley, 2004.
- [7] M. Powell and J. Swann. "Weighted Uniform Sampling—a Monte-Carlo Technique for Reducing Variance". *Applied Mathematics*, Volume 2, Number 3 (September 1966), pp. 228–236.
- [8] T. Ritschel, T. Grosch, and H.—P. Seidel. "Approximating Dynamic Global Illumination in Image Space". *Proceedings ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, 2009, pp. 75–82.
- [9] P. Shanmugam and O. Arikan. "Hardware accelerated ambient occlusion techniques on GPUs". *Proceedings of the 2007 Symposium on Interactive 3D Graphics*, 2007, pp. 73–80.
- [10] T. Umenhoffer, B. Tóth, and L. Szirmay-Kalos. "Efficient Methods for Ambient Lighting". *Spring Conference on Computer Graphics*, 2009, pp. 99–106.

- [11] S. Zhukov, A. Iones, and G. Kronin. "An ambient light illumination model". *Proceedings of the Eurographics Rendering Workshop*, 1998, pp. 45–56.

19

Real-Time Character Dismemberment

Aurelio Reis

id Software

Overview

Modern games utilize a number of tricks to convey a realistic and intriguing world. In only a few years, realistic physics and destructible environments have garnered widespread industry adoption, yet despite this, few games attempt to model heavy damage on game characters. One of the main reasons for this is the complexity of decomposing the topology of a 3D mesh dynamically with real-time performance.

In this gem, we introduce an efficient general-use technique for character dismemberment that can be easily incorporated into an existing skeletal animation system. This implementation is perfectly suitable for games, real-time applications like military simulations, and other "serious games" where the accuracy of the damage modeling does not need to be precise (as in medical simulations).

While many games don't have the subject matter appropriate for dismemberment, when presented in a cartoonish way it's possible to approach gore and dismemberment such that it is relevant to the gameplay experience as opposed to being used purely for shock value or gimmick. Movies such as *Evil Dead 2* and *Kill Bill* have approached gore in a comedic way that rivets audiences. As zombie and monster games gain popularity,

it's likely that they will have the most to gain from character dismemberment.

19.1 What is Character Damage Modeling?

A character in this context is roughly defined as any animated figure in the shape of a creature using a bipedal skeleton and keyframed animations for its motion. In the game world, a character is represented as an entity that is capable of moving around using some scripted or autonomous behaviors with animations that coincide with this motion. As the character moves throughout the world, it may come in contact with a number of interactive elements that can affect the character in one way or another. In a first-person shooter, for instance, a character may move about, be influenced in some way by the environment (like when riding an elevator), and may shoot or be shot by other characters (player or non-player).

When a character is killed, the results are usually presented using some kind of pre-canned animation sequence with special effects like blood and gore strewn about. In most modern games, it is common practice to use "ragdoll" physics once the character is completely dead to add an additional sense of realism. In addition, some games go so far as to create bloody chunks and body parts, so-called "gibs" (for giblets). Combined, these elements result in character damage modeling that makes the game world more believable to the player. Games, after all, are about consequences, and realistic character deaths add to that realism.

While most games are able to get away with this degree of damage modeling, some games require more precise detail. Additional elements like projected decals and character scarring adds a lot, but the most dramatic change would be in the shape of the character itself. The effects of severe blunt force or impact trauma on a humanoid is enough to dismember most major limbs with ease, and it is this particular subset of damage modeling on which this gem focuses.

19.2 Methods of Mutilation

There are a number of ways to approach character dismemberment. The easiest and most straightforward solution would be to explicitly model the character as an articulated collection of body parts. Whenever a limb's bounding volume is hit, everything below that body part in the hierarchy can be manually detached. This technique, while incredibly simple, can be difficult to tune because model seams become prominent at the body part boundaries. Aesthetically, the quality is quite poor, although it can be hidden by clever modeling tricks (e.g., placing a gorget at the character's neck seam). In addition, this technique would require a large number of body parts to work well, which means additional draw calls that can result in reduced performance.

The extreme opposite of this would be to use computational geometry. Constructive solid geometry (CSG) boolean operations can be used to remove chunks of a character by dynamically splitting and removing polygons from the mesh. The uniform character mesh is a great advantage, but unfortunately this technique requires heavy CPU processing both in pre-transforming the animated mesh to the proper pose and in the required geometric operations. This technique can also result in a highly triangulated mesh with an unpredictable number of polygons that can become both a memory and performance concern.

Another possible method would be similar to the one used in the *Soldier of Fortune* series of games made by Raven Software. In their approach, they used what they called "gore zones" to represent up to 26 areas that can be toggled off on a given animated biped. The character models were created in such a way that every gore zone was completely capped, sealed and internally textured. This allowed them the flexibility to represent internal cavities such as the skull where the brain could become detached due

to heavy trauma. The obvious downsides are a heavily triangulated model and a draw call for each gore zone—26 draw calls per character model. This can be avoided by consolidating gore zone geometry into a massive draw batch, but since this requires touching GPU memory, it can come at an expensive cost.

It's also possible to remove limbs by using geometry morphing. In this method, a blend shape is used to move the vertices of the limb to a desired final position, i.e., a stump or cap. This technique is easily hardware accelerated, although older hardware is only able to handle a few blend shapes. Because of this, dismembering multiple limbs at the same time requires modifying a vertex buffer where a morph scale is specified. In order to determine whether to apply the vertex position deltas to get the dismembered geometry location, this scale is modified at run time. Like the previous technique, this method suffers from having to modify a buffer in GPU memory.

Ultimately, each of these techniques has its own set of potential benefits and pitfalls. Ideally, what we want is a system that works with a uniform mesh because of the inherent performance benefits of a single draw call. In addition, our solution should be able to take advantage of commodity graphics hardware like that in the current generation of game consoles. This system also needs to be flexible enough to allow for any number of user-defined body parts (within reason) to be severed, generating a new and separate object that is able to coexist with the original model. Finally, it should require as little artist intervention as possible, allowing for arbitrary break points based on user-defined data that is easily modifiable.

19.3 Bone Matrix Flattening

The solution presented in this gem works like so. Given a 3D mesh and a matching skeletal hierarchy, a user-defined damage zone is created for every breakable body part. Each damage zone contains a bounding volume that defines its area, the joint to which

it is attached, the surface area it encompasses (more on this later), and the list of joints below the main joint (see Listing 19.1). The bounding volume should be as tight-fitting as possible. An oriented bounding box works quite well since most limbs on a human biped are longer than they are wide although a sphere may also be sufficient (depending on the underlying geometry). Figure 19.1 shows an example of a damage zone configuration.



Figure 19.1: The limb damage zones.

Listing 19.1: The damage zone class definition.

```
class CArDamageZone : public CArHitBox
{
public:

    CArDamageSurface    m_Surface;

    // The next damage zone below this one in the skeletal hierarchy.
    CArDamageZone      *m_Next;
```

```
// The joints below this damage zone's joint
vector<int>      ChildJointList;

// The largest joint index in the limb hierarchy.
int             m_JointRange;

C_ArDamageZone() : m_Next( NULL ), m_iJointRange(INVALID_JOINT) {}
~C_ArDamageZone() {}

void GatherChildJoints(const SMD5Skeleton& Skel);
};
```

When a hit is registered on a body part's damage zone, every child joint below the joint to which it is attached is traversed, and each one of these joints is moved to the position of the main joint. In addition to this, each matrix for the child joints is flattened, that is, scaled down to zero to form a degenerate matrix. This effectively creates a stump as the vertices reduce to a singular point.

The process for creating the dismembered piece is very similar. The same damage zone joint is now used as the origin of all the joints above it in the hierarchy and their transforms are also flattened as in the previous step. The detached limb can now function as a separate entity where it can come under control of the physics system, i.e., go into ragdoll mode.

19.4 Improvements

While this technique's results can be utilized immediately, there are a number of things that can be improved. First, while the end caps formed by the matrix flattening don't look that bad, they don't really convey any kind of localized damage. To remedy this, it's possible to use "gore caps" and "blood flowers". These artist created models can

be attached to the end of a damage zone joint to create the appearance of bone shards or bloody entrails. Since they are defined per damage zone, it is possible to provide a unique context-specific gore cap for any given break point. It's also possible to detect whether a matrix has been flattened within a shader and react accordingly, possibly blending between a damage texture created with procedural texture coordinates or fading out the polygons completely to show interior surfaces. In coordination with particle systems for effects like blood, a very rich visual presentation can be created.



Figure 19.2: A limb is removed by transforming the vertices associated with its joints by a skinning matrix that makes those vertices degenerate. The detached limb is formed by performing the process in reverse.

Another area of improvement has to do with how detached limbs are rendered. While degenerate triangles that have no pixels generated save on fill rate, the vertices still have been processed. This has the potential to waste quite a bit of vertex throughput on the GPU. To fix this, we can scan all the vertices in the model and store the ones that belong to the child joints of a particular damage zone as a continuous "damage surface". The generated surfaces can then be recombined to form a new mesh that has its vertices

prearranged by limb order. When it comes time to render a detached limb, only the range of vertices and triangle indices defined for a given damage zone need to be rendered. The different color coded damage zone surfaces are shown in Figure 19.3.



Figure 19.3: (See also Color Plates.) The color coded damage zone surface groups.

It's possible to skip this decomposition step by prearranging all the model geometry to be contained within a mesh for each damage surface. In this way, the only processing needed would be to recombine the individual meshes into one uniform mesh in order to gain the benefits of a single render batch.

While this technique works well for drawing isolated triangles for detached limbs, it does not work for the base character model, as any number of possible limbs can be broken off. This means it's impractical to create vertex group configurations that could

accommodate rendering of only specific subsets of the model without adding additional draw batches. Also, the limb break points don't need to lie at the origin of a damage zone joint as was described earlier. By means of projecting the actual hit location along a line connecting the damage zone joint and its immediate child, it is possible to create a break that is more precise. Another enhancement worth exploring would be adding mirrored joints in the model's skeleton to allow rendering of the main character body and the dismembered limb in the same draw call.

Lastly, while the technique described here was described in the context of a bipedal creature, it can be used for any object with discrete "limbs", such as tree (branches), light poles, or even non-bipedal creatures.

19.5 Demo

A demo of the technique described in this gem can be found on the accompanying CD. The MD5 format was chosen for the model and animations since they cover the bases on most skeletal animation needs. The core logic for the damage zone surface generation can be found in the `CARBaseModel::GenerateDamageSurfaces()` function in `Model.cpp`. The logic and definition of the damage zone is in `BoundingBoxVolume.h/.cpp`. The logic that calculates the animated skeleton along with the flattened joints for the limb and body is in the `CARAnimator::Update()` function in `Animator.cpp`.

20

A Deferred Decal Rendering Technique

Jan Krassnigg

University of Aachen, Germany

Overview

Rendering decals is a common method used to apply more detail to 3D worlds dynamically. Decals are often used to add bullet holes, blood stains, tire marks, and similar items to a world as events occur in a game, but they can also be used by level designers to enrich the environment with wear and tear textures, dirt textures, signs on walls, etc.

This gem presents a decal rendering technique that uses deferred shading to produce scenes like the one shown in Figure 20.1. The technique is entirely shader based, extremely lightweight on the CPU, does not need to dynamically generate triangles, and is a straightforward addition to most 3D engines.



Figure 20.1: Decals applied to complex geometry.

20.1 The Problem

There are several things that a good decal system should solve:

1. The system should integrate well with lighting. Decals should not only change the color, but also change the normal, specular factors, and any other surface parameters such that they become indistinguishable from all other geometry.
2. Decals should work convincingly on all surfaces, static and dynamic.
3. Decals need to clip properly to geometry boundaries, and possibly even wrap around corners.
4. The system needs to work with geometry that might be very different from the geometry used for collision detection.

Item 1 in this list is easily solvable when the graphics engine includes deferred rendering capabilities [2]. For forward renderers, the system in this gem can still be used, but some modifications will be necessary. For a thorough explanation of deferred shading, please refer to [3,4,5,6].

Item 2 can be solved if we can get our hands on a free 8-bit channel in the G-buffer. If not, we can at least make it work on static geometry. This is discussed in Section 20.5.

Item 3 is where the real problems begin. The decal needs to follow the surface to which it is applied, even if that surface is highly tessellated. Some existing decal rendering techniques generate a triangle representation for a decal on such a surface [1], but this can be computationally expensive. Furthermore, the raw mesh is often not available at all on the CPU since the mesh data is loaded into a vertex buffer accessible only by the GPU at a reasonable speed.

Item 4 is an issue because today's games often use very complex meshes for rendering, but a less detailed mesh might be used for collision detection. The difficulty is that the point of intersection that a ray cast returns might differ quite a bit from the location where the decal must be rendered.

20.2 The General Idea

The basic solution afforded by our technique is to project a decal onto a surface using a special fragment shader applied to the decal's bounding volume instead of rendering new decal polygons on top of the scene geometry. The only information required to achieve this consists of the position and orientation of the decal, the decal's size, and a texture containing depth values for the viewport being rendered. This information allows all computation to be done in the vertex and fragment shaders.

When rendering a decal, it is natural for us to work in "decal space", where the x -

and y -axes lie in the tangent plane to the underlying surface at the decal's center, and the z -axis is parallel to the surface's normal vector at the decal's center. In order to move data into the decal's local coordinate system, the shader needs to be supplied with the inverse of the decal's transformation matrix.

The code in Listing 20.1 first fetches the depth from the G-buffer at the fragment position and uses it to reconstruct the 3D world-space position of the fragment. The `worldToDecal` constant is the inverse of the transformation matrix for the decal that we are currently rendering. With this matrix, we can transform the fragment's position into the local space of the decal. This local position can then be used to compute texture coordinates at which the decal texture is sampled. In this first version, we are using the (x,y) position only, which simply projects the decal along its local z -axis onto the underlying geometry, as shown in Figure 20.2.

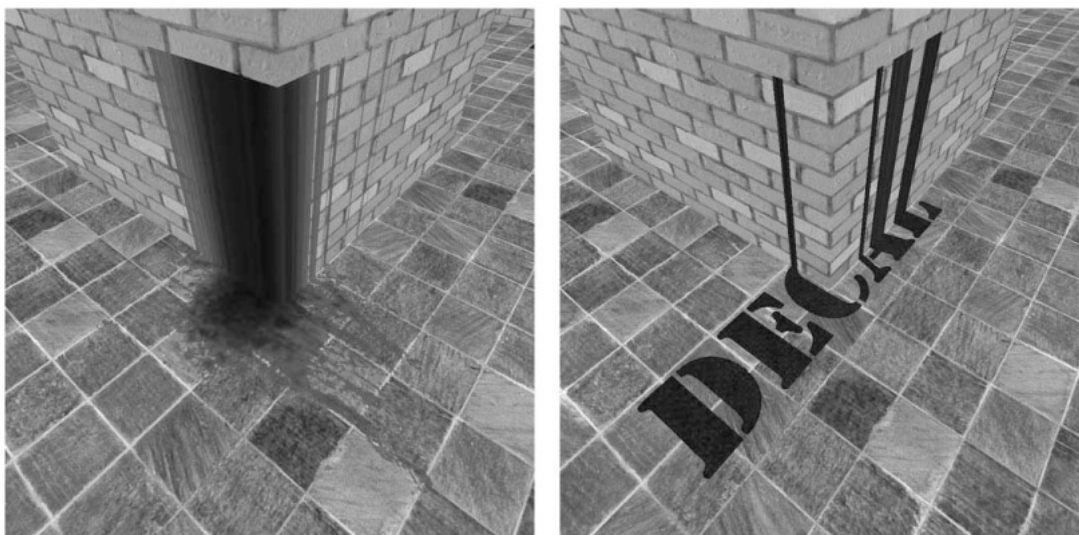


Figure 20.2: Using a simple projection onto the local x - y plane causes decals to be smeared in the direction of the local z -axis.

Listing 20.1: This code determines the decal-space coordinates for the fragment being rendered and transforms them into texture coordinates for the decal. The `RT_Depth` texture contains depth values for the viewport, the `worldToDecal` constant is the inverse of the decal's 4×4 matrix transform from decal space to world space, and the `recipDecalSize` constant is $1/s$, where s is the size of the decal in the scene.

```
// Sample the depth at the current fragment
float pixelDepth = texture2DRect(RT_Depth, gl_FragCoord.xy).x;

// Compute the fragment's world-space position
vec3 worldPos = ComputeWorldSpacePosition(gl_FragCoord.xy, pixelDepth);

// Transform into decal space
vec3 decalPos = worldToDecal * worldPos;

// Use the xy position of the position for the texture lookup
vec2 texcoord = decalPos.xy * recipDecalSize * 0.5 + 0.5;

// Fetch the decal texture color
gl_FragColor = texture2D(diffuseDecalTexture, texcoord);
```

20.3 Geometry Rendering

Now that we have some code that calculates basic texture coordinates, we must consider what kind of geometry we actually need to render. Each fragment rendered with the decal shader acts like a "window" through which we can possibly see our decal. So we could just render a surface-aligned quad corresponding to the size and position of the decal. However, we want our decal to have depth, so we instead render a bounding cube as shown in Figure 20.3. This allows us to see the decal from any direction, and it will allow us to add a wrap-around feature later on.

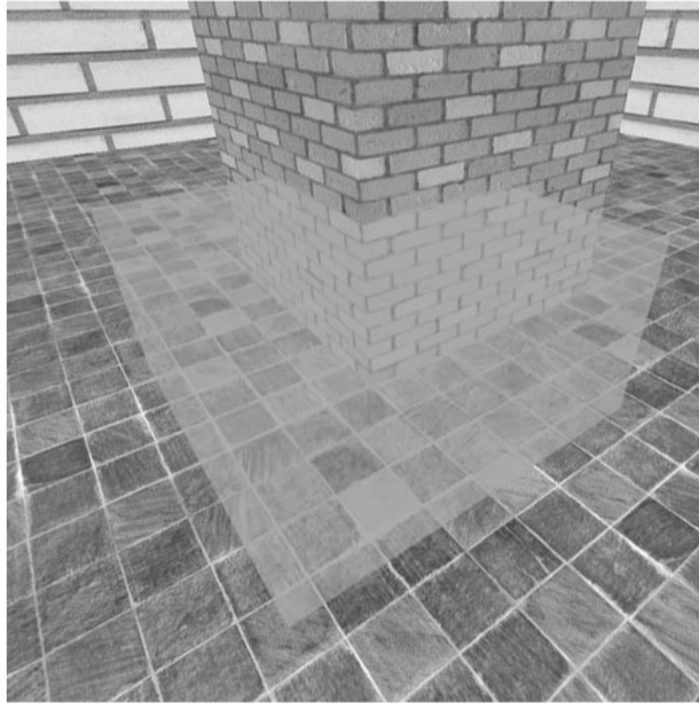


Figure 20.3: A bounding cube centered on a decal can be rendered to ensure that we capture the decal's influence from any viewpoint.

As an optimization for scenes containing a large number of decals, it can be advantageous to render decals through hardware instancing [7]. Therefore, it is a good idea to simply render a unit cube and transform it to the correct size and position in the vertex shader, as demonstrated in Listing 20.2. Note that we can easily extend the uniform constants `decalSize` and `decalToWorld` to arrays and use `gl_InstanceID` to render a batch of decals through instancing.

Listing 20.2: This vertex shader scales a unit cube to the actual size of the decal and translates it to the decal's world-space position.

```
uniform float   decalSize;  
uniform mat4    decalToWorld;  
  
// scale the unit cube and position it in world space  
vec4 worldPos = decalToWorld *  
    vec4(gl_Vertex.xyz * decalSize, gl_Vertex.w);  
  
// output the position in homogeneous clip space  
gl_Position = gl_ModelViewProjectionMatrix * worldPos;
```

For small decals, this technique works very well. However, when a cube is rendered with backface culling enabled and the depth test set to `GL_LESS`, the decal disappears the moment the camera enters the cube. One solution to this problem is to cull front faces and set the depth test to `GL_GREATER`. This way, only fragments whose world positions are actually behind surfaces are rendered, but it causes many fragments to be rendered unnecessarily when they are occluded by geometry closer to the camera.

Another solution is to find the corner of the cube that is closest to the camera and render a camera-aligned quad at that depth that is large enough to enclose the entire cube. If the closest corner is in front of the near plane, a full-screen quad should be rendered at the near plane instead.

20.4 Fade Out And Wrap-Around

We have a basic vertex shader and fragment shader in place, but our projection is infinite along the decal's *z*-axis, producing the smearing shown in Figure 20.2. There are two methods we can use to fix this problem. The simpler method is to use the distance from the fragment's position to the decal plane as a fade-out parameter. This distance is already available in `decalPos.z`, and we just have to scale it to the size of the decal as demonstrated in Listing 20.3.

Listing 20.3: The absolute value of the decal-space z-coordinate of the fragment position is scaled to the size of the decal and used as a fade-out parameter for the decal color. As before, the `recipDecalSize` constant is $1/s$, where s is the size of the decal in the scene.

```
// compute the distance of the fragment to the decal's plane
float distance = abs(decalPos.z);

// scale the distance into the [0,1] range
// according to the size of the decal
float scaledDistance = max(distance * recipDecalSize * 2.0, 1.0);

// somehow use that scaled distance to fade out
// here: simple linear fade out
float fadeOut = 1.0 - scaledDistance;

vec4 diffuseColor = texture2D(diffuseDecalTexture, texcoord);

gl_FragColor = vec4(diffuseColor.rgb, diffuseColor.a * fadeOut);
```

This method is useful when a decal is supposed to be flat without wrapping around geometry. The exact formula used to fade out a decal can be varied to produce the best look for different decals, and it is advisable to make this configurable within the engine.

A more interesting method for handling the smearing problem is to make a decal wrap around corners and follow the curvature of complex surfaces. This is especially useful for blood stains and other liquids that splatter because it is much more convincing if such a decal covers an entire surface independently of its curvature.

This is quite easy to achieve. All we need is the surface normal at the position of each fragment in the decal. If we rotate that normal into decal space, its (x, y) components give us the gradient of the surface relative to the decal plane. We can use this gradient and the fragment's distance to the decal plane to modify the texture coordinates. In areas with no relative slope, the texture lookup remains unchanged, but

in areas with a large slope (for example, at corners), the texture coordinates move outward according to the distance to the decal plane. This technique is illustrated in Listing 20.4, and the result is shown in Figure 20.4.

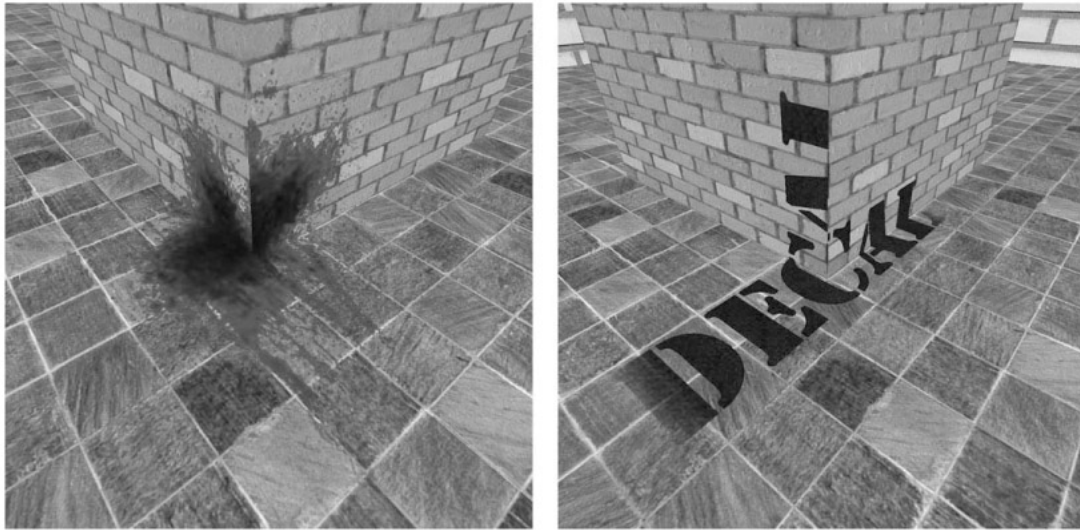


Figure 20.4: The decals wrap around corners based on the surface normals, and they fade out based on the distance from the decal plane.

Listing 20.4: This code demonstrates how the normal of the underlying surface can be used to adjust texture coordinates in such a way that a decal wraps around curves and corners. The `RT_Normal` texture contains normal vectors for the viewport encoded in the RGB channels.

```
// get the world-space normal at the fragment position
vec3 worldNormal = texture2DRect(RT_Normal, gl_FragCoord.xy).xyz;

// rotate it into the local space of the decal
vec3 decalNormal = (worldToDecal * vec4(worldNormal.xyz, 0.0)).xyz;

vec2 texcoord = decalPos.xy;
```

```
// use the distance and gradient to modify the texture lookup
texcoord -= decalPos.z * decalNormal.xy;

// scale and center the texture coordinates
texcoord += vec2(decalSize);
texcoord *= recipDecalSize * 0.5;

gl_FragColor = texture2D(diffuseDecalTexture, texcoord);
```

For the finishing touch, we add color fading that depends on the distance to the decal plane. We also need to account for the angle between the decal plane's normal and the underlying surface normal, since otherwise, decals appear on the back side of thin walls. A complete example shader with more details can be found on the accompanying CD.

20.5 Surface Clipping

The technique we have described works by applying decals in viewport space and does not differentiate among surfaces onto which it is projected. By using the projection and wrap-around method, it is possible to attach decals to any kind of surface, even to animated characters. However, a decal is never clipped, meaning that a decal attached to a box also projects onto the ground on which the box is resting. If the box moves and carries the decal along with it, then the projection on the ground moves as well.

There are two possible solutions to this problem. One solution is to restrict decal rendering to pixels covered by static geometry only. In this case, we need to render all static geometry first, then apply decals, and only afterward, render dynamic objects.

The second solution requires that an additional channel in the G-buffer be used to hold a "decal ID" for every distinct object that is rendered. In the rendering pass that fills the G-buffer, we write each object's decal ID along with the diffuse color, normal,

etc., so that we have a per-pixel mask identifying to which object each pixel belongs. When a decal is applied to an object, we look up the object's ID and associate it with the decal. When the decal is rendered, we pass this ID along as an additional uniform constant and compare it to the ID read from the G-buffer. If the two match, then we know that the pixel belongs to the object to which the decal is attached, and we render normally. Otherwise, we discard the fragment because it would be drawn outside the set of pixels covered by the object.

If no distinction needs to be made among static geometries, then they can all share the same ID, such as zero. All dynamic objects should have different IDs, but those IDs don't need to be unique. All we need to do is make it very improbable that two dynamic objects near each other have the same ID. It then suffices to use an 8-bit channel and simply give the dynamic objects random IDs from the range [1,255].

This kind of object ID management could also be done using the stencil buffer. However, testing stencil values prevents us from using instancing to render the decals due to the fact that we cannot pass the stencil compare value as a uniform constant to the shader and change the stencil test on a per-fragment basis.

20.6 Limitations

There are a few limitations one should be aware of. This technique does not magically create volumetric decals. It only improves a 2D projection so that it looks more realistic in many situations. The wrap-around feature uses the normal of the underlying surface to change the texture coordinates inside a decal. The results shown in Figure 20.4 look much better than in Figure 20.2, but there is still a clearly visible cut at the edge of the vertical column. To prevent such artifacts, one would need to use truly volumetric decals. Figure 20.5 shows a decal applied to a more complex object. In the left image, the object uses face normals, and in the right image, it uses smooth normals.

In both images, no normal mapping is considered. Obviously, the normal of the underlying surface has a big effect on the appearance of the decal. Consequently, surfaces with strong normal mapping tend to distort the decal, sometimes causing visual artifacts. For some kinds of decals, this is less of an issue, though, because a blood stain usually still looks like a blood stain no matter how distorted it becomes when applied to a surface.

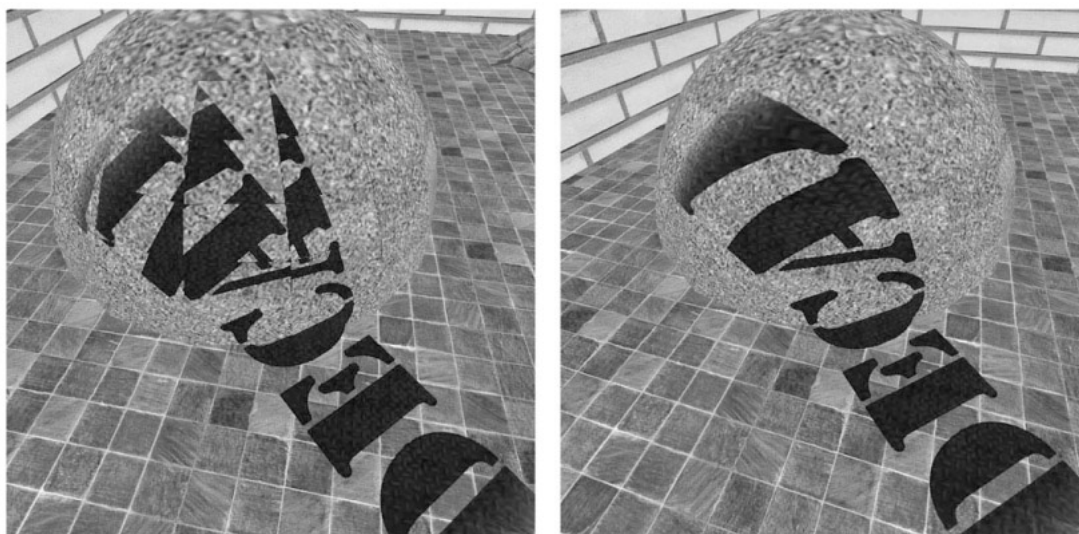


Figure 20.5: (Left) Wrap-around based on face normals. (Right) Wrap-around based on smoothly interpolated vertex normals.

A word about performance: It is easy to cull decals by testing bounding volumes, and the GPU can render decals in large batches through instancing. However, you should be careful not to have many decals layered on top of each other because doing so can easily consume all available fragment-rendering power. You can try to prevent such situations by removing decals after a certain amount of time or by restricting the number of decals on screen at once. The latter approach allows us to keep many decals

in the world as long as they are not visible at the same time. Most games simply clean up the scene by removing decals after a fixed amount of time, but lose a lot of immersive quality in the process. The game *Max Payne* places a limit on the number of decals created in a single area, which has the great effect that if the player comes back to a room where he was previously, all the blood stains and bullet holes are still there.

20.7 Additional Features

Since we are already rendering the decals into the G-buffer before any lighting occurs, we can not only change the diffuse color, but can also replace (or modify) the normal, gloss, and other data as we wish. Computing the tangent and bitangent vectors for the decal is very straightforward in the vertex shader, and no additional vertex attributes are needed. For more information, see the example shader on the accompanying CD.

References

- [1] Eric Lengyel. "Applying Decals to Arbitrary Surfaces". *Game Programming Gems 2*, Charles River Media, 2001.
- [2] Joris Mans and Dmitry Andreev. "An Advanced Decal System". *Game Programming Gems 7*. Charles River Media, 2008.
- [3] Oles Shishkovtsov. "Deferred Shading in S.T.A.L.K.E.R.". *GPU Gems 2*. Addison-Wesley, 2005.
- [4] Frank Puig Placeres. "Fast Per-Pixel Lighting with Many Lights". *Game Programming Gems 6*. Charles River Media, 2006.
- [5] Rusty Koonce. "Deferred Shading in Tabula Rasa". *GPU Gems 3*. Addison-Wesley, 2008.

[6] Dean Calver. "Deferred Lighting on PS 3.0 with High Dynamic Range". *ShaderX3*. Charles River Media, 2005.

[7] Francesco Carucci. "Inside Geometry Instancing". *GPU Gems 2*. Addison-Wesley, 2005.

Part III



Programming Methods

21

Multithreaded Object Models

Jon Parise

Electronic Arts

Overview

Many simulation games are constructed around a core model in which everything in the game world is represented by an object. Each object is a data container, maintaining self-consistent state for an in-game entity. Objects can be defined using a type hierarchy or a composition pattern, and they can be organized spatially (as in a scene graph) or in flat collections.

One challenge common to all of these implementation schemes is how to represent consistent object state in a multithreaded environment. For example, a game engine could update the simulation state on one thread, render the game on a second thread, and perform animation and physics work on additional threads. Data synchronization, consistency, and conflict resolution quickly become problems.

This gem discusses four approaches to solving these problems:

- Explicit locking
- Message-based updates
- Multiple thread contexts
- Buffered state changes

Each approach varies in its complexity, flexibility, and performance. To that end, there is no one perfect solution for all game engines, but the information provided in this article should facilitate choosing the most appropriate architecture to match a known set of requirements.

21.1 Explicit Locking

The most common approach to multithreaded data contention is to add explicit locking using operating system synchronization primitives, such as critical sections and mutex locks. Locks are generally used to protect concurrent access to shared, centralized data resources, such as message queues or device state, and for these types of use cases, they are a good way to ensure data consistency across multiple threads. Synchronization primitives are generally well-supported, the interlocked scopes are explicit, and the initial implementation cost tends to be low.

While locks are often easy to implement, they should be used with care. First, locks introduce blocking behavior into a multithreaded application, and the potential for blocking (or worse, deadlock) increases with the number of locks in the system. Second, many synchronization primitives consume operating system resources (such as kernel handles under Windows). Third, there is no point at which an object's state (or multiple objects' states) can be considered consistent. Last, it is generally difficult to verify the correctness of large lock-based systems without using advanced analysis tools and frequent code audits.

In the specific case of object models, each object could potentially require one or more locks in order to support multithreaded access. Depending on the size of the game's simulation, this could result in a large amount of overhead, potentially compromising game performance. Given that, the fine-grained, explicit locking approach seldom scales to large, complex simulation games.

21.2 Message-Based Updates

A message queue can be used to serialize updates from multiple "writing" threads. Each message in the queue describes an object-related state change. The queue is protected by a single lock that each writer must acquire in order to write a message. A single "reading" thread periodically processes the queue's contents by acquiring the lock, playing back the individual messages, and applying the state updates to their associated game objects.

In this model, all object modifications ultimately occur from the single processing thread, so protecting individual objects becomes unnecessary. Reading object data from multiple threads is still problematic, however, because there are no data consistency guarantees for threads other than the one performing the message processing.

There are three measurable forms of overhead associated with this approach. The first involves the memory overhead of the message queue itself. Second, state changes are delayed for as long as their associated message is in the queue waiting to be processed. And third, all writing threads must block while waiting to acquire the queue's lock, which could be held by other writers or by the processing thread. The processing thread does not necessarily need to block if it fails to acquire the lock immediately, but waiting until its next attempt will further delay the enqueued updates, and there is still no guarantee it will be able to acquire the lock without blocking.

Given these considerations, a message-based approach is most appropriate for engines that distribute work from a single primary thread to multiple worker threads. It provides a nice mechanism for serializing the results of those jobs and then applying them in the context of the primary update thread. Synchronization is limited to a single lock protecting the message queue, which reduces complexity and aids debugging.

21.3 Multiple Thread Contexts

Another solution explicitly identifies the object data fields that need to be accessed concurrently. These fields are duplicated and partitioned into per-thread context structures. Each thread effectively gets its own "private" version of the fields that it can access without using locking primitives, enabling asynchronous reading and writing. The fields' contexts are periodically synchronized to update all threads' views of the data.

Methods that access these fields select the appropriate context structure using the current thread ID. If a method modifies the field's value, it must also make a record of the change, such as updating the field's assigned bit in the object's per-thread "modified fields mask". This allows individual changes to be tracked, which drive the synchronization process.

At the end of each frame, all modifications are merged: the changes made by one thread are copied to the other thread's context, and vice-versa. This process can be optimized by using a per-thread "change list" to track unresolved objects (in conjunction with the "modified fields mask" mentioned above).

In the event that both threads have modified the same field independently, the merge is ambiguous. In these cases, one thread is considered authoritative, and the opposing thread's modified value is overwritten. In a properly organized system, however, this should rarely, if ever, happen. Most fields will only be modified by one thread while being read by many.

The merge operation is performed by an explicit synchronization step at the end of a rendering frame (at the vertical blank, for example) when both threads enter a common barrier. This effectively ties the non-rendering threads to the rendering thread's update rate, but ideally the other threads are performing time-sliced work that is compatible with this timing model. Alternatively, synchronization could be based on

a non-rendering thread's update rate (such as simulation updates), but then the rendering thread could not be locked to a specific visual refresh schedule.

This architecture can be extended to support additional per-thread context structures containing thread-local fields that are not synchronized. This allows thread-specific code to store object-specific data using the existing object system. For example, the rendering system could store scene culling information on each object that would only be accessible through that thread's context.

The memory cost associated with this approach scales with the number of duplicated contexts. The minimum number of contexts is two, so this architecture could potentially halve the number of objects that can fit in memory at one time, assuming all object fields needed to be synchronized. There are also runtime costs associated with selecting the active thread, tracking modified fields, and merging contexts.

This approach has the advantage that all thread synchronization is centralized within the object system. Code that works with objects can remain ignorant of the underlying synchronization system and will never block outside the explicit synchronization barrier, which is predictable. It also guarantees a consistent view across all object data from each thread's perspective, as long as all data access respects the per-thread context partitioning scheme.

21.4 Buffered State Changes

The final approach presented here is fairly strict and complex, but it also has a lot to offer in terms of features and architectural cleanliness. The implementation centers on a dedicated command queue that buffers multiple frames of state changes from one thread to another. This effectively splits the engine into two halves: a dedicated

simulation thread, which produces the state changes, and a rendering thread, which presents the simulation state to the player.

Another way to think about this separation is in networking terms: the simulation thread acts like a network server, the command queue fulfills the role of the network transport layer, and the rendering thread is similar to a network client application that presents the buffered state changes.

The command queue is designed to hold multiple frames of simulation state. The simplest possible implementation uses two frames in a traditional double-buffering scheme: while one frame is being written by the simulation thread, the other frame is presented by the rendering thread.

Frame swaps are controlled by a single lock. The simulation thread holds the lock while it writes its state into a new frame, forcing the rendering thread to continue rendering its current frame until the new frame is made available by the simulation. Rendering is therefore never stalled by the simulation, but it is forced to represent the same simulation state over multiple rendering frames when it is running at a faster update rate than the simulation.

Because it is quite common for the renderer to run at a faster rate, a better command queue implementation uses three or more frames of simulation state. This layout allows the renderer to consider multiple frames of simulation state, interpolating between them based on timing information embedded within the frames. Each additional frame of data in the queue adds latency and memory overhead, however, so the ideal layout for many engines uses three frames.

In order to facilitate interpolation, each frame includes a timestamp indicating when the data was written by the simulation. Also, each entity within a frame, such as an object, is assigned a unique handle. When an entity is serialized, both its current handle and its handle from the previous frame are written. This allows the renderer's

interpolation logic to match an entry in one frame to its corresponding entry in the previous frame. Handles are generally represented as indexes or offsets into the frame's buffer.

In addition to buffered, interpolative state, the command queue can also handle discrete events. Frame events are triggered by the simulation and have some associated visual effect, such as starting or stopping a particle system. When one of these events is written by the simulation, it becomes associated with the simulation's current frame. Buffered events won't be executed by the renderer until it is also ready to evaluate that frame's serialized state as well. This rule maintains visual consistency between object state changes and events.

There are three clear advantages to using a frame-based dependency queue. First, it results in a clean separation between the simulation and rendering systems. Second, communication between simulation and rendering is one-way, removing any potential for ambiguity concerning the state of the world. Lastly, timing and thread synchronization are explicit, allowing the threads to run at different update rates.

This type of architecture does carry some heavy costs, however, particularly in terms of memory usage. The size of each frame in the queue can be significant, and, depending on the game data and serialization requirements, frame sizes might vary from one to the next, making it difficult to anticipate the peak memory size required by the queue. There are run-time processing costs associated with serializing, unpacking, and interpolating buffered state, and there is the cost of delaying state changes while they're buffered in the queue. Lastly, in terms of overall engine design, separating systems into rendering and simulation "halves" can be a challenge.

21.5 Selecting the Best Approach

Four different approaches to synchronizing object state in a multithreaded game engine have been presented. All of them have been used successfully in many shipping titles, but selecting the most appropriate architecture for a specific engine requires careful evaluation of that game's requirements. In addition, it is entirely possible to implement hybrid schemes, perhaps on a per-system basis, but care must be taken to avoid introducing additional data consistency problems as a result.

It's important to emphasize the weight that engine and gameplay requirements should have in this decision. For example, if the game embeds a scripting language, the implementation details of the scripting system may dictate additional requirements with regard to how script interacts with the game's object system. The scripting environment could even host the entire object system itself, further influencing how the object system is connected to the rest of the game engine. Alternatively, the gameplay may not tolerate input delays, in which case some of the buffering-based approaches discussed above may introduce unacceptable latency.

It's also important to consider external libraries when designing this aspect of the game engine's architecture. Many middleware packages have specific requirements with regard to data access patterns. For example, DirectX on the PC requires that multithreading support be explicitly enabled, and it comes with a measurable performance penalty, so engines that target the PC platform generally restrict rendering operations to the main application thread.

With hardware and software trends clearly embracing parallel execution environments, dealing with concurrent data access problems is becoming an increasingly common problem. Game engines must be designed with both runtime and developer efficiency in mind, and the concurrency challenges put even more emphasis

on establishing solid architectural foundations. Hopefully, the material presented in this gem has contributed some useful techniques to that aspect of game engine development

22

Holistic Task Parallelism for Common Game Architecture Patterns

Brad Werth

Intel Corporation

22.1 Tasks Versus Threads in Games

Parallel programming in games has typically relied upon the use of threads to provide concurrent execution of work. Threads are independent processing streams that are given execution time based upon a scheduling algorithm in the host operating system. This system works well as long as there are enough hardware resources (CPU cores) to run the available threads. However, modern games run on a variety of platforms, with very different core topologies. Predetermining a specific number of threads for a fixed amount of work is no longer a viable strategy.

A popular alternative strategy is to define a thread pool, with a number of threads scaled to fit the available hardware resources. Parallel work is divided into tasks and assigned to the threads for execution. These tasks differ from threads in that they should never block while waiting on other computation. When the task is started, it is run to completion and cannot be swapped out for the purpose of running another task. So the challenge of coordinating the activity of multiple threads is changed into the effort of

determining how to divide work into tasks and under what conditions those tasks should be run. The second half of this gem demonstrates methods for breaking down typical parallel game patterns of work into tasks.

Figure 22.1(a) shows an example of how some sample work can be divided into tasks. The total work has an initial period of uninterrupted work *A*, but then needs to wait on the occurrence of some event *E* before proceeding with additional work *B*. This dependency on the event *E* effectively splits this section of work into two tasks, *A* and *B*. Later, work *C* is a section that could be split into pieces and run concurrently (a data decomposition). If it is split into pieces, work *D* must wait for all of those pieces to complete before it can proceed. This means that the rest of the work can be split into some number of tasks C_1, C_2, \dots, C_n and the remaining task *D*. The final task breakdown is shown in Figure 22.1(b).

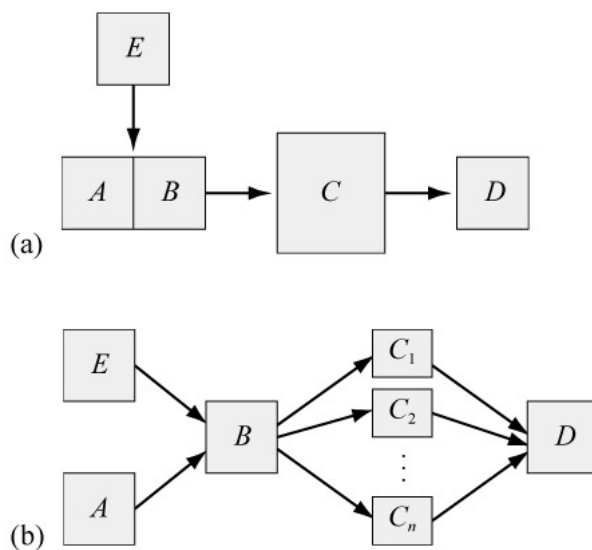


Figure 22.1: (a) Work to be divided into tasks. (b) The work expressed as dependent tasks.

Taken alone, this transformation from threads to tasks can seem underwhelming.

The real power of this approach becomes evident when a game architecture is able to leverage large amounts of parallel work at once. In that case, the tasks can be packed into the threads in the pool with very few gaps of inactivity. Mapping all the tasks to the thread pool in an efficient fashion is the responsibility of a task scheduler, the other focus of this gem.

22.2 The Task Scheduler

There are only a few features that must be implemented in a task scheduler. At a minimum, it must be possible to dispatch and wait on tasks. Dispatched tasks are assigned to a thread in the thread pool, the size of which can be specified at initialization time. Although this can be done simply with a single shared task queue, most task scheduler implementations have a more complicated internal architecture for performance reasons.

The key performance improvement is the use of per-thread task queues. This eliminates the synchronization chokepoint when one shared task queue is used. If a task spawns additional tasks, the new tasks are added to the current thread's queue. This introduces the possibility of queue size imbalance, which is typically resolved by the use of *work stealing*. Work stealing allows a thread that has emptied its own task queue to take work from another thread's queue. Advanced task schedulers may use heuristics to determine which thread to steal from and which task to steal, and this may help cache performance. Together, these improvements eliminate significant synchronization overhead in the task scheduler.

Even though the minimal feature set is so simple, it can be daunting to create a task scheduler from scratch because of the difficulty of writing correct multithreaded code. Thankfully, there are existing examples to use or study: Nulstein [1] is a small and simple free source task scheduler for Windows, and Intel Threading Building Blocks [2]

is a highly optimized scheduler with both a proprietary and an open source license available for Windows, OS X, and Linux. The open source version has also been ported to the Xbox 360. Intel Threading Building Blocks is used for the code samples in this gem, but any task scheduler will work as long as it has the minimum feature set: dispatching and waiting on tasks in a scalable thread pool.

22.3 Decomposing Game Patterns into Tasks

A task scheduler can efficiently execute tasks, but parallel games don't typically use tasks directly. Instead, there are a number of patterns that have emerged in game architectures for handling parallel work. Conveniently, it's not difficult to transform these patterns into task-based patterns. These transformations are described for those creating a task scheduler as well as for those working with an existing task scheduler. The patterns can be implemented in either case, but it is often more efficient to extend the task scheduler to support the patterns that your game actually uses.

The examples below describe each pattern and show an excerpt of the complete source code that accompanies this gem on the CD. You will need to examine the complete source code to see the whole pattern in action.

Callbacks and Futures

Games frequently have a need to run some work in parallel to the main work. This parallel work is sometimes structured as a function that sets a flag when complete. This flag can be checked later to determine when the parallel work has finished. This is an example of a *callback* system, and it maps directly into a task-based pattern by using the `dispatch` method of the task scheduler:

```
TaskManager::JobResult result;  
bool flag = false;  
taskManager->dispatch(&result, (TaskManager::JobFunction) doCallback,  
                      &flag);
```

The obvious improvement that can be made to this system is to allow the dispatching thread to wait on the completion of the function pointer. This is called the *future* pattern and is supported by the task scheduler's dispatch and wait methods:

```
taskManager->dispatch(&result, (TaskManager::JobFunction) doFuture,  
                     NULL);  
taskManager->wait(&result);
```

A well-designed task scheduler will ensure that waiting on a task is an active wait; instead of sleeping, the "waiting" thread will execute parallel work if there is work available. The full code for this example is "Callback and Future Sample" on the accompanying CD.

Independent Loops and Splittable Tasks

Loops appear frequently in game architectures, and some of those loops contain large amounts of computation. If the iterations of a loop are logically independent and the number of iterations is known at the start of the loop, then subsets of iterations can be grouped into a task. This is typically done by defining the body of a loop as a function that takes a single parameter, the context object. The context object encapsulates all the data needed by the original loop, which includes at least the start and end indices over which to iterate. The task scheduler is given a function pointer and an array of context objects that have been initialized with index subranges to cover the original loop's iteration range. The task scheduler method prototype looks like this:

```
void dispatchMultiple
(
    JobResult *result,      // structure to track completion
    JobFunction func,       // pointer to task function
    void *params,          // array of context objects for tasks
    size_t paramSize,      // size of one context object
    unsigned int count      // number of tasks to create
);
```

Although this approach is effective, it requires the game code to allocate an array of context objects, to decide ahead of time how many tasks to split the loop into, and to initialize the context objects with the appropriate subranges. To avoid these constraints, the task scheduler can provide a method to dispatch splittable tasks. A splittable task has a function pointer with three parameters: in addition to the context object, it is also passed the start and end indices. The scheduler passes the parameters to the task's function pointer, but it must also determine if a task's index range should be split into subranges assigned to subtasks. An additional function pointer is needed to determine whether and how to split a range in two pieces. With all of these elements in place, the game code can transform an independent loop into tasks without allocating arrays of context objects or predetermining the number of tasks to create. The task scheduler method prototype looks like the following. (The full code for this example is "Loop Sample" on the accompanying CD.)

```
void dispatchSplittable(
    JobResult *result,      // structure used to track completion
    JobRangeFunction rangeFunc, // pointer to task function
    JobSplitFunction splitFunc, // function that splits a range
    void *param,           // context object for tasks
    unsigned int start,     // beginning of range
    unsigned int end        // end of range
);
```

Long, Low-Priority Operations

Games occasionally rely on computation that needs to run continuously, but not at the expense of other more immediate work. Level loading, asset decompression, and AI pathfinding are common examples. The low-priority operation runs continuously and provides periodic output to the main work of the game architecture. On its surface, this seems like a pattern that is fundamentally at odds with task parallelism. However, if the continuous computation can be changed into an iterative algorithm, then each iteration of the algorithm can be treated as a task.

Once the transformation is complete, the next challenge is to ensure that these tasks are given a low priority relative to other tasks in the task scheduler. Task schedulers can handle this in a few ways: the task queues for each thread can be changed into priority heaps, or the low-priority task can be inserted into the task queue in a location that makes it less likely to be run immediately (an implementation-specific detail). If you are using an existing task scheduler without this capability, you can dispatch low-priority tasks opportunistically when other tasks are being dispatched:

[illegible]

```
}

// Now we dispatch whatever we were asked to dispatch.
taskManager->dispatch(result, func, param);
}
```

This workaround is effective since most task schedulers assign tasks to threads in last-in-first-out order. If the most recently added task spawns additional parallel work, tasks inserted earlier will generally not be started until the new parallel work has been started. The full code for this example is "Low-Priority Sample" on the accompanying CD.

Synchronized Callbacks

Occasionally, games need to do per-thread initialization before running parallel work. This is useful when interacting with some threaded middleware packages. When you create the task pool directly, it is trivial to ensure that each thread makes the necessary calls. But when your thread pool is managed by a task scheduler, it can be more complicated. As long as your task scheduler uses work stealing, you can dispatch a number of tasks equal to the number of threads in the pool, and use synchronization primitives to prevent those tasks from completing until all have been assigned to threads:

```
tbb::task *TaskManager::SynchronizedTask::execute()
{
    ASSERT(m_func != NULL);
    m_func(m_param);
    (*m_atomicCount)--;

    while (*m_atomicCount > 0)
    {
        // yield while waiting
        TaskManager::yield();
    }
}
```



```
}  
  
    return NULL;  
}
```

If your task scheduler does not use work stealing and does not allow you to directly assign tasks to threads, then another option is to define your tasks to do a just-in-time check for per-thread initialization when they are run. The full code for this example is "Synchronized Sample" on the accompanying CD.

Directed Acyclic Graphs

Earlier in this gem, we looked at how an arbitrary piece of parallel work could be split into tasks, resulting in a directed acyclic graph (see Figure 22.1). Many games conceive of their parallel work this way and would like to submit tasks to a task scheduler preceded by a list of ancestors. The trick to implementing this pattern is to encapsulate the tasks in objects that can manage the dependency relationships and actually dispatch the tasks when appropriate. When a task completes, it notifies the dependency-tracking objects, which may trigger more tasks being dispatched to the task scheduler as follows. (The full code for this example is "Graph Sample" on the accompanying CD.)

```
void doDAGTaskFunction()  
{  
    // First, we call our function pointer.  
    m_func(m_param);  
  
    // Now we tell our children that we're done.  
    tbb::concurrent_vector<DAGTask *>::iterator it = m_children.begin();  
    while (it != m_children.end())  
    {  
        (*it)->parentDone(this);  
    }  
}
```

```
    it++;  
}  
}
```

22.4 The Future of Task Parallelism in Games

Established habits of game development have been disrupted by the installed base of multi-core CPUs. In the same way, game development will be disrupted again by the introduction and evolution of powerful many-core processors that require special programming techniques in order to be used effectively. With the techniques discussed in this gem, you'll be able to create task parallelism abstractions that will get your game running quickly in any parallel hardware environment.

References

- [1] Jérôme Muffat-Méridol. "Do-it-yourself Game Task Scheduling". Intel Software Network, 2009. <http://software.intel.com/en-us/articles/do-it-yourself-game-task-scheduling/>
- [2] Intel Threading Building Blocks. <http://www.threadingbuildingblocks.org/>

23

Dynamic Code Execution Hierarchies

Martin Linklater

Sony Studio Liverpool

Overview

As hardware becomes more complex and powerful, the software that they run becomes larger and more complex. As software grows, the number of programmers needed to write and maintain the software increases. As anyone who has experienced large programming teams will tell you, the more programmers you have working on one product, the more time you need to spend policing the code structure and managing its complexity.

As different programmers create and modify game systems, it is easy for the code to become a mismatch of different patterns and styles. Programmers also tend to have their own personal preferences when it comes to object creation, initialization, updating, and deletion.

Inter-object communication, if not tightly regulated, can become a spaghetti of code dependencies and object graph traversals. As programmers build objects that require data from other objects, it is all too easy to create monolithic dependency graphs and be forced to navigate awkward and non-intuitive object linkages. It is also too easy to create header file dependencies that have nothing to do with the actual job you are

trying to accomplish, but are required so you can connect to other objects in the code.

Keeping track of code construction can become a full-time job, and modifying the code during the latter stages of a project can become error prone and dangerous. Fragile code is difficult to bug-fix without causing more bugs.

This gem discusses code execution hierarchies (CEH) as a way of controlling these problems and visualizing your code when things get confusing.

23.1 What are Code Execution Hierarchies?

Code execution hierarchies provide a framework for code construction and updating. You use a simple base class for all of your major game components, and define the update order by way of a manager class with an internal tree structure. Rather than create objects and add them manually to the main loop, as in Figure 23.1(a), you create your objects and attach them to the execution tree relative to an already existing object. Conceptually, the structure is similar to a scene graph in graphics programming where each object has a parent, and the object's location in the scene is always defined relative to its parent. When you move or update a node in the graph, all of its children are automatically moved with it, maintaining the parent-child relationships. The main loop using a code execution hierarchy is shown in Figure 23.1(b).

Explicitly defining the parent-child and sibling relationships of your code modules not only forces people to *think* before adding an object to the update system (always a good thing), but also helps protect your code from structural changes in the future. If a code module needs to be updated somewhere else during the frame, you move the parent, and the child objects are automatically moved along with it.

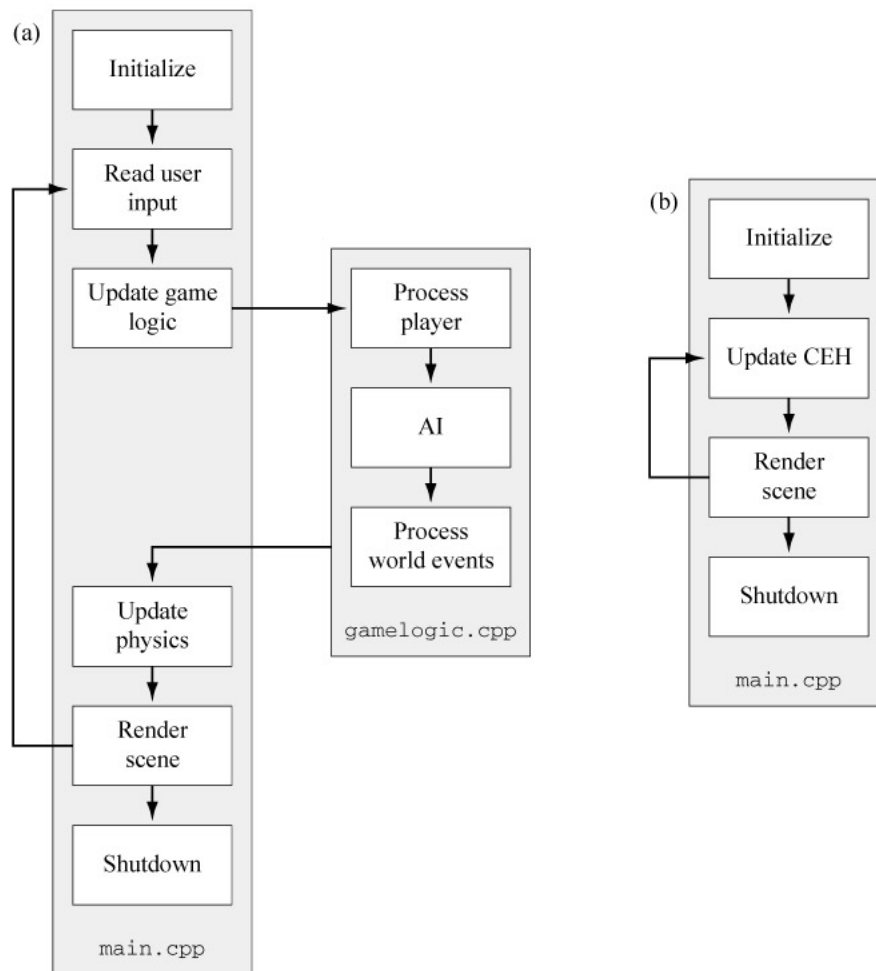


Figure 23.1: (a) Traditional update loop. (b) Code execution hierarchy.

A simple base class is used for execution objects so that the manager class has a consistent interface for your objects. This base class also defines the API for object creation, initialization, updating, and deletion. Runtime type identification systems provide a simple way for objects to identify themselves within the hierarchy. This is used when objects need to find objects of a certain type within the hierarchy. A simple

base class may look like that shown in Listing 23.1. Objects can be created and added to the hierarchy as shown in Listing 23.2. Rather than building your code explicitly and defining your update order in the main loop, you are creating code objects and inserting them into the code execution hierarchy.

Listing 23.1: Example base class.

```
class CEHBase
{
public:

    CEHBase();
    virtual ~CEHBase();

    virtual void Init(void);
    virtual void Update(float dt);
    virtual void ... grab the latest code and prune the good bits

protected:

    m_classID;
};
```

Listing 23.2: Example game code.

```
MyObj *obj = new MyObj;
obj->Insert(kAsChildOf, kRootNode);
obj->Init();

OtherObj *other = new OtherObj;
other->Insert(kAsSiblingAfter, obj);
other->Init();
```

23.2 Design Features

Tree Structure

The object hierarchy itself is a tree structure. There is one root node, owned by the manager class. Each object has one and only one parent. Each object has zero or more siblings, and each object can have one child link. Once you have your tree structure, there are two relationships that are required to be maintained when parsing the tree:

- Parents are always updated before their children.
- Siblings are always updated in the same order, first to last.

Given these two relationships, there are two ways to traverse the tree and update your objects: depth first and breadth first. Which one you choose depends on your requirements, but due to memory access patterns and performance I favor the breadth-first traversal. Figures 23.2 and 23.3 show a simple graph and the update orders for both systems. As you can see, the two rules above are maintained for both traversal systems—previous siblings are processed before next siblings, and parents are processed before children. A real-world example of a CEH might look like that shown in Figure 23.4.

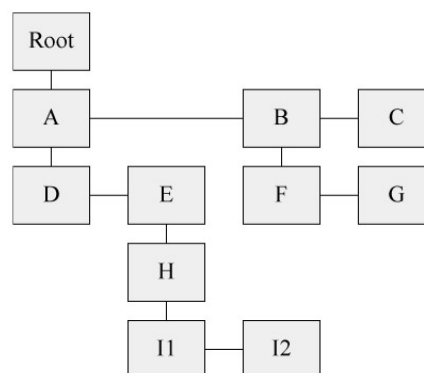


Figure 23.2: Simple graph.

Depth first



Breadth first



Figure 23.3: Update orders.

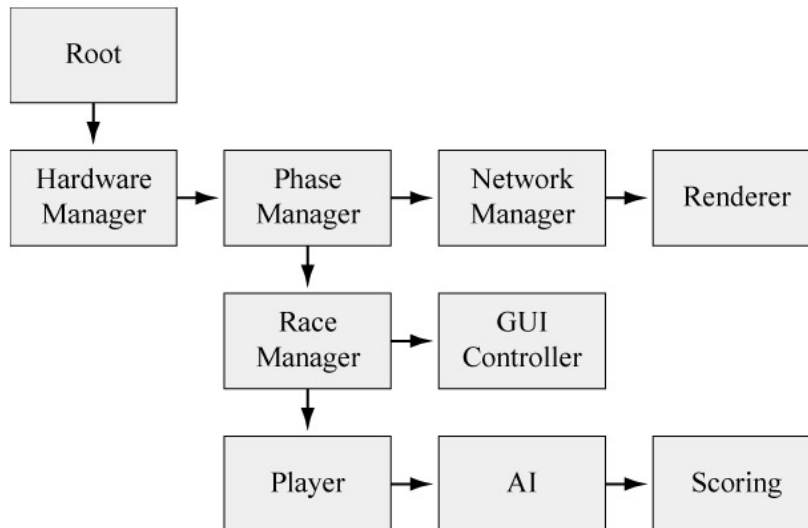


Figure 23.4: Example CEH Graph.

Time Deltas

You may have noticed that the `Update()` method in the base takes a floating-point `dt` parameter. This is used to tell your object how much time has elapsed since the last call to `Update()`. Making your code flexible regarding update frequencies and decoupling objects from V-sync events can have great benefits, as long as you are willing

to do the extra work to make your object internals robust for varying `dt` values.

Adding awareness of time also gives your CEH the potential to deal with varying timing requirements for its objects. For instance, it is simple to build basic timer functionality into the manager class, allowing you to specify how often various objects require updating. If you have an object that only needs to be updated approximately once each second, then that object can tell the manager that its desired update interval is only every second. The manager class can then handle the update call for you.

Dynamic Structure

Since the update hierarchy is built at run time, it can be treated as a dynamic data structure. Update order is not hardwired into your code, but can be modified and altered as required. If network code does not need to be updated due to there being no network connection, you simply omit it from the update graph. This can be much cleaner than placing the following `if` statements all over your code.

```
if (networkActive)
{
    ...blah...
}
```

It also allows you to define your update graph via a data file, allowing runtime behavior to change without recompiling your code.

Introspection

By building an introspection system into your CEH base class you allow for much more flexible code construction. I'm sure we've all been in the position of needing to get a handle on a certain class instance and being forced to navigate a lot of run-time linkage to get at the object. The alternative is to make the desired object global, rarely a good thing. As an example, in Figure 23.2, for object E to get a pointer for object G, it could

be required that E goes through code like this:

```
ptrG = GetD()->GetA()->GetB()->GetF()->GetG();
```

This is rather clunky, and if the linkage of intermediate classes changes, say objects A and B are swapped, all of the code that manually navigates the linkages needs to be updated.

Using a CEH with introspection, the call could look like this:

```
ptrG = GetFirst<G>();
```

This code does not need to be altered if the hierarchy changes since the CEH manager does all the work of locating the first instance of class G and returning a pointer to it. This is how using CEHs can make your code more resilient to code restructuring. The introspection system can also provide APIs that deal with vectors of objects and filtering of objects based on relative location (above, below, etc.). Object linkage is no longer explicit in code, but dynamic at run time.

Visualization

Visualizing the execution of your code can be a great help in debugging and bringing new members of your team up to speed with your code. Using a CEH can help make this visualization consistent and useful. Since the calling graph and actual call order are controlled by your manager class, you can get the manager to output data about your execution behavior for you. A handy format that I use is to output the execution graph in DOT file format [1], and then run this file through GraphViz to get a snapshot of how the code is linked and executed.

Deferred Operations

There are certain operations that are very dangerous to perform while you are mid-way through the graph traversal. Any operation that modifies the graph (delete, move, create) during the update phase can have disastrous effects since you are altering the

same tree you are traversing. To shield against this, dangerous operations can be queued until the end of the update phase. These deferred housekeeping tasks can catch you out if you are expecting the modifications to happen immediately, but the added layer of bug protection it gives you is worth the trouble.

23.3 Benefits & Pitfalls

Moving your execution control over to a data driven, dynamic system like code execution hierarchies has both benefits and pitfalls. The benefits include:

- Code is constructed with a consistent pattern (Init, Update, Destroy, etc.).
- Update order is easier to modify since the order is dynamic rather than hardcoded.
- Introspection can make finding object instances easier than manually navigating class linkages.
- Introspection allows the layout of objects to change without the need to maintain manual linkage code.
- It is simple to visualize the execution order and hierarchy of your code. This aids debugging and teaching of programmers new to the team.

There are, of course, a few pitfalls and caveats that come with the use of code execution hierarchies. These include:

- It is difficult to see what is happening by just looking at the code. Since the update order is dynamic and built at run time, manual inspection of the source code doesn't help much.
- Some code simply doesn't fit. Even though CEHs provide a simple and flexible structure, there are still pieces of functionality that don't fit into the framework. You still have to deal with these systems manually and take care of object linkage by hand.

- You need to be careful with your granularity. Putting game systems and substantial game objects into the CEH can be very useful. Placing every particle in a particle system is overkill, and you will waste lots of CPU cycles and memory to unnecessary CEH overhead. You need to use your judgement.
- Modifying the execution tree while you are traversing it can cause lots of catastrophic but difficult-to-find bugs. You need to be very careful when altering the tree or defer all modification operations until after the update phase has completed.

Code execution hierarchies are a huge topic for discussion. I have worked with code that uses explicit execution order, linear lists or queues of update functions, and deeply embedded tree hierarchies, and they all have their good and bad points. My hope is that this gem has introduced a new way of thinking about code execution to those who have not used dynamic systems before and has possibly provided some food for thought for those currently using dynamic execution systems. Game code inevitably becomes complex, and new ways of looking at how we construct code are always useful. Using consistent and flexible methods like those I have described can help ease the pain and provide much needed abstractions.

References

- [1] DOT file format. <http://www.graphviz.org/doc/info/lang.html>
- [2] GraphViz. <http://www.graphviz.org/>

24

Key-Value Dictionary

Martin Linklater

Sony Studio Liverpool

Overview

This gems presents a design for a flexible, observable repository for game configuration data. The key-value dictionary (KVD) is a data structure inspired by the key-value observing [1] technology used in Mac OS X Cocoa frameworks. When used sensibly, the KVD can simplify your code.

The KVD is designed to be flexible, easy to use, and have few external dependencies. It is not designed to be blisteringly fast, or to be used for frequently modified data. The KVD is well suited for storing game state information that is read often, but modified rarely. The KVD code on the accompanying CD is written in C++ and uses the standard template library (STL) for internal storage containers. The KVD is lightweight (about 350 lines of C++) and simple to integrate into an existing game engine.

24.1 Design

Apple's OS X relies heavily on Objective-C and the Cocoa frameworks. One of the

fundamental mechanisms of Cocoa that binds the frameworks together is called "key-value observing" (KVO). Apple describes KVO as follows:

Key-value observing provides a mechanism that allows objects to be notified of changes to specific properties of other objects. [1]

KVO is a mechanism that, once you get used to it, becomes almost invaluable. Code does not have to repeatedly poll a value to detect whether it has changed; rather, the code registers its interest in learning when a value changes and is notified when the value does change. The code is notified of the new value by way of a callback. Since Apple's KVO mechanism requires Objective-C and Cocoa, but we write games primarily in C++, I have created the KVD, a simple C++ data repository that mimics simple KVO behavior.

The KVD can take data of any type, with each piece of data having multiple observers. Observers are notified of changes immediately upon the value changing. The KVD also checks that a value has actually changed before sending notifications, so setting a variable to the same value it currently holds will not trigger any notifications.

24.2 Using the KVD

As mentioned in the introduction, the KVD is designed to be used with general game state data that is read often, but changed relatively infrequently. Traditionally, game state data is repeatedly polled by lots of different systems. For example, the current screen resolution is a piece of data that changes very infrequently, but which is polled by a number of different systems. If you use the KVD to store the screen resolution, you don't need to poll for its value each frame, but you are instead told when the value changes and what the new value is.

Using an intermediary data store like the KVD can help decouple your classes and

reduce header file dependencies. As in the previous example, if you were to store the current screen resolution inside the rendering system, every piece of code which is required to read or set this value needs to include the header file for the rendering system, and that could itself pull in other headers as illustrated in Figure 24.1, increasing compile times.

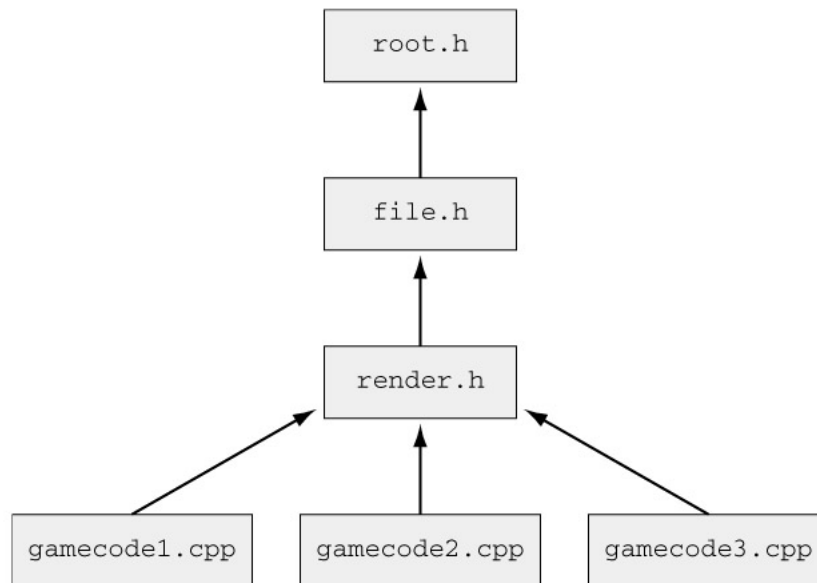


Figure 24.1: A polling model for accessing information often creates additional header dependencies.

Pulling the screen resolution out of the rendering system and into an intermediary would stop your game code from having to include the rendering system header (and all the headers the rendering header includes), but would add a dependency on the intermediary. As long as this intermediary header has fewer dependencies, you have simplified the header file dependency chain as shown in Figure 24.2, and this will speed up compile times.

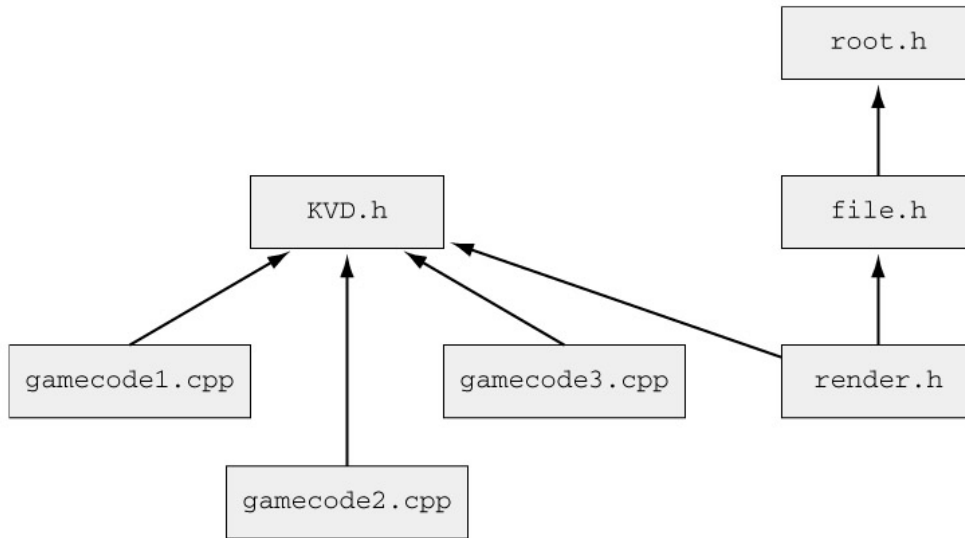


Figure 24.2: The header dependency graph is simplified by moving information into an intermediary header.

24.3 Code Details

The key-value dictionary has a relatively small API and only has external dependencies on the STL. To use the KVD, you first need to create a KVD object by declaring it as follows:

```
KeyValueDictionary myKVD;
```

Once created, the KVD is ready to accept values and notifications. Setting a value in the KVD is as simple as this:

```
myKVD.Set<int>(std::string("myInteger"), 1);
```

```
myKVD.Set<MyStruct>(std::string("myStruct"), myStructInstance);
```


Notifications happen through a function callback mechanism. The callback functions take the following form:

```
void NotificationFunc(void *newValue, void *userData)
{
    // react to new value
}
```

To add a notification to the KVD, you need to tell the KVD which function to call when the value changes (the first parameter), and which value you want to watch (the second parameter). The third parameter is optional and will be passed to the callback function as the `userData` parameter. This can be set to null or set to a pointer to your own user data associated with the callback notification.

```
myKVD.AddNotification(NotificationFunc, std::string("myInteger"), 0);
```

The callback function is passed two values: the first is a pointer to the new value, and the second is a pointer to the user data that was set when the notification was added. Since the callback isn't told the type of the data, you have to cast the `newValue` pointer to the appropriate type. You have to make sure that the type used is consistent for a given key. Mixing types can cause difficult-to-find bugs or crashes.

Once you have set a notification, whenever the value is changed using the `Set()` method, your notification is called. You can add however many notifications you want to each key value—they will all be called whenever the value changes. Notifications can be removed using the `RemoveNotification()` method:

```
myKVD.RemoveNotification(NotificationFunc, std::string("myInteger"));
```

You can get the value for a key manually, if you so wish, with the following call:

```
int    myValue;

myKVD.Get<int>(std::string("myInteger"), &myValue);
```

Internally, the primary storage container for the KVD is an STL map. Each map element is indexed by the key string hash, and contains the value encoded as a `std::string`, a lock, and a `std::list` of notification callbacks. Each notification callback contains a pointer to the callback function, and the user data value, as illustrated in Figure 24.3.

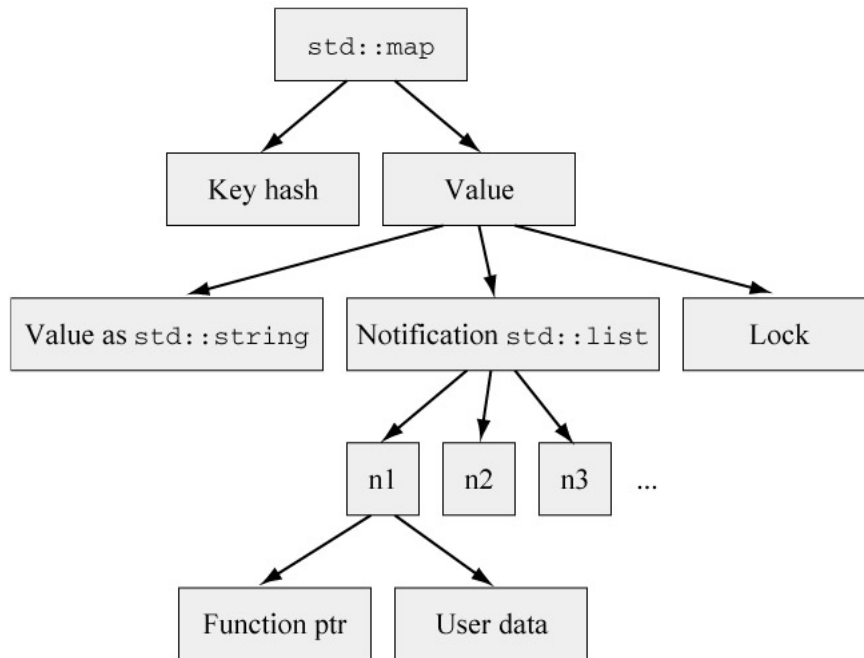


Figure 24.3: The data stored in a KVD.

STL was chosen due to its good performance characteristics, common availability, and robust nature. If your implementation of KVD requires different container characteristics, you are free to change the source to suit your needs.

Each entry in the KVD map has a lock flag. Whenever a key value changes, the lock is set before the notification phase begins and cleared after notifications have completed. This lock is checked *before* a key value is modified, to make sure that recursive change

notifications don't happen. Recursive change notifications are a bad thing because they can blow the stack and crash your program. Consider the case where a key is altered within the call graph of its notification function. Each change in value would trigger a notification, which would change the value, triggering the notification, etc. The locking mechanism stops this from happening. Since locks are present for each individual entry in the KVD, however, you are allowed to alter a different KVD entry from inside a callback.

24.4 Caveats

The code presented in this article is not perfect and is meant as a starting point for you to modify as you see fit. There are a number of issues that I have not tackled on purpose, since requirements will differ among uses:

- **Thread safety.** The code is not thread-safe. If you require thread safety, you will need to add whatever your mechanism of choice is to the code. If your game code is quite traditional and keeps all the KVD logic on one thread, you can ignore this issue.
- **Performance.** Although the KVD is reasonably fast, you may have specific performance requirements affecting the details of which containers and mechanisms are used. For general use cases though, the code should perform adequately as is.
- **Memory allocation.** The internal storage for the KVD is handled via the STL default allocator. This could cause fragmentation issues in your code. If you need to keep a firm handle on memory usage, you will need to either write your own container code or override the default allocator for the STL containers that are used.

References

[1] "Key-Value Observing Programming Guide".
<http://developer.apple.com/mac/library/DOCUMENTATION/Cocoa/Conceptual/KeyValueObserving/Concepts/Overview.html>

25

A Basic Scheduler

John Bolton

Netflix

Highlights

Many games have a need to execute tasks at regular intervals. Threads are a possible solution, but they can be a poor choice due to their nondeterministic nature, the complexities of their interactions, and the high overhead of context switching. On the other hand, a scheduler can be implemented to run in a single thread and execute tasks at a specific point in the frame. Its tasks will execute in a single context under full control of the application.

This gem presents a basic lightweight object-oriented scheduler that implements limited cooperative multitasking between tasks in a single thread. Possible applications of this scheduler include AI, audio, and environmental effects. The complete code for the scheduler described here is available on the accompanying CD.

25.1 Overview

The scheduler implemented here is designed to execute a list of tasks one at a time at a particular point in a frame or time step. Each task has a timer maintained by the

scheduler, and all tasks whose timers have expired are queued to be executed in that frame. Tasks are executed serially by the scheduler in the scheduler's thread, and each task runs to completion. Thus, tasks are never interrupted by the scheduler or by other tasks, and there is no need for synchronization among tasks.

Order of execution within a frame is arbitrary in this implementation, but the order can be controlled by implementing a priority system. Other additional features, such as load balancing, are not implemented here, but are described later.

25.2 Task Functionality

The task as seen by the scheduler is very simple. There is an initialization function, a cleanup function, and a function to execute the task. To provide the interface for this functionality, tasks are derived from the base class `Task`, which has the virtual functions `Start()`, `Stop()`, and `Execute()`.

The `Start()` function is called immediately after the task is added to the scheduler's task list. Its purpose is to allow the task to initialize itself before it begins any execution. It is safe (as far as the scheduler is concerned) to remove the task from the scheduler's task list in the `Start()` function.

The `Stop()` function is called immediately after the task is removed from the task list. Its purpose is to allow the task to clean itself up. Once the `Stop()` function is called, the scheduler no longer references the task, so it is safe (as far as the scheduler is concerned) to destroy the task anytime during or after the call to the `Stop()` function. It is also safe to re-add the task to the scheduler's task list from inside the `Stop()` function.

The `Execute()` function is called periodically by the scheduler according to the task's period. The timer is maintained by the scheduler. When a task is finished executing, it returns one of the following values that the scheduler uses to manage the

task:

- **ACTIVE.** If this result is reported, the task is requeued for execution again according to its period. This is the normal result.
- **AGAIN.** If this result is reported, the task is requeued to run again in the next frame, regardless of its period. This result is intended to indicate that the task could not finish successfully and should be executed again as soon as possible. It could also be used to force the task to be executed every frame, but it is better to set the period appropriately instead.
- **INACTIVE.** If this result is reported, the task is removed from the scheduler's task list. The scheduler removes all references to the task and then calls the task's `Stop()` function.

25.3 Scheduler Functionality

The scheduler class maintains a list of tasks and schedules them for execution according to their periods. There can be more than one instance of the scheduler class, though for the sake of simplicity, instances cannot be copied or assigned in this implementation, and tasks do not keep track of which scheduler is executing them. In order to determine when tasks are to be executed, the scheduler maintains a timer for each class. Each frame, the timers are updated, and all tasks whose timers have expired are executed in an arbitrary order. The scheduler also provides the ability to suspend and resume executions of tasks and to change the period of a task.

Tasks are added to a scheduler's execution list by the `Add()` function. The `Add()` function also specifies how often the task is executed. If the task being added is already in the scheduler's task list, an error is returned. Tasks may be added at any time, including while tasks are being executed. The task's timer is initialized when the task is added. As described above, a task's `Start()` function is called after it is added.

Tasks are removed from the scheduler's execution list by the `Remove()` function. Tasks can be removed at any time. If the task being removed is not in the scheduler's task list, an error is returned. Tasks are not executed once they are removed, even if they are removed during a frame in which they are scheduled to be executed. As described above, a task's `Stop()` function is called after it is removed.

When the `ExecuteTasks()` function is called, all tasks that are ready to run are executed serially in an arbitrary order. The `deltaTime` parameter to the `ExecuteTasks()` function indicates the amount of time that has passed since the last time the function was called. This value is used to update the tasks' timers in order to determine when tasks become ready to be executed. The `ExecuteTasks()` function is intended to be called once per frame, but it is possible to call as often as desired. As long as the `deltaTime` variable is accurate or at least reasonable, the tasks will execute with the intended period. The exceptions are tasks with a period of `PERIOD_EVERY_FRAME` and tasks that return the value `AGAIN`. These tasks are executed whenever `ExecuteTasks()` is called.

Tasks are suspended and resumed using the `Suspend()` and `Resume()` functions. Suspended tasks are not executed and their timers are not updated. If the task being suspended or resumed is not in the scheduler's task list, an error is returned. This differs from `Add()` and `Remove()` in that the task remains in the scheduler's task list, the task's `Start()` and `Stop()` functions are not called, and the task's time is frozen until `Resume()` is called.

The period of a task can be changed by the `SetPeriod()` function. If the task is not in the scheduler's task list, `PERIOD_INVALID` is returned. When a task's period is changed, its timer is reset. A period of `PERIOD_EVERY_FRAME` causes the task to execute every frame. Calling `SetPeriod()` with an invalid period (less than 0), does not change the period and returns the value `PERIOD_INVALID`.

25.4 Implementation

The data structure used by the scheduler is very straightforward. A pointer to each task along with a timer and some state information is stored in a vector. An STL container is chosen in order to simplify the implementation. A fixed-size array would alleviate memory allocation issues, but then additional logic would be necessary in order to prevent overflow.

Other container types might be considered depending on how the scheduler is used. The tasks are not sorted because it is assumed that sorting the tasks would be more time-consuming than simply scanning all entries to find tasks to execute. If there are a very large number of tasks and only a few are executed each frame, then it might pay to sort the tasks.

Task execution is performed in three phases. First, the timer for each task is decremented according to the amount of time that has passed. Then, for each task, if the timer is less than or equal to zero, the task is executed (unless it is suspended or marked for removal), and the timer is reset to its period. Finally, all tasks marked for removal are removed. The purpose of the three phases is to avoid problems that might arise when one task modifies another task, or adds or removes tasks from the scheduler.

25.5 Additional Functionality

Some additional features are not implemented here for the sake of simplicity. In this implementation, tasks are run during each frame in an arbitrary order. It might be advantageous to give tasks a priority so that they can be executed in a certain order with respect to the other tasks that are executed in the same frame. In order to accomplish this, the priorities of the tasks are stored in the task list and the tasks can be sorted by

priority or stored in a data structure that supports a priority scheme.

In some situations, the amount of time available for executing tasks might be limited or budgeted. In this case, the `ExecuteTask()` function could have an additional parameter specifying the budgeted time, and the scheduler would execute tasks until the budget is used up. The task's `Execute()` function could have an additional parameter specifying the time remaining, allowing the task itself to limit the amount of time it uses. Budgeting time could also work in conjunction with task priorities, reducing the latency of higher priority tasks. However, it must be noted that this simple scheduling algorithm can become inadequate in some situations.

26

The Game State Observer Pattern

Ron Barbosa

Revelex Corporation

Overview

With today's high-powered graphics and audio hardware, developing a game is becoming more akin to producing a blockbuster movie. Real-time ragdoll physics are the new stunt men, and particle systems are the new pyrotechnics. With so many sexy components required to bring a game engine together, it's not surprising that little treatment is given to the management of game state.

When the user clicks the mouse button or presses the left analog stick of a gamepad, the state of the game is what determines whether the user intended to fire his avatar's weapon or select the "Quit" option from the menu. The game state determines whether the graphics hardware should render the game screen or the inventory menu. Whether NPCs should execute their next animation frame or just wait around for the next game loop iteration is due in large part to the game's state.

Game state management can be made streamlined and elegant using an implementation of the *observer design pattern*.^[1] The observer pattern provides a way for instances of classes (subjects) to be "observed" by other objects (observers) in the application. Each observer subscribes to the subject, and when the subject's data is

changed, it sends notification to all registered observers, providing each subscriber a reference to the subject. The observers can then query any public data or call any public methods of the subject to determine what has changed and how it affects the observer.

Combining the observer pattern with a game state manager would allow the game state manager to notify all interested software components of what the game state is in real time. So if the user hits the "Pause" option, the game state can be set to paused. The state manager would then notify the avatar manager that the game is paused, and it can stop processing controller input until further notice. The state manager would simultaneously notify the menu manager that the game is paused, and the menu manager could begin rendering the pause menu and responding to controller input to select menu options.

The real-time notification mechanism simplifies the code by encapsulating the effect of state change on an individual module in the software. Without a proper state management mechanism, developers will often use arbitrary or artificial conditions to determine what should be done during the game loop. Consider the following pseudocode for an avatar's game loop processor.

```
public void Update([arguments])
{
    // Determine if the input should be processed
    if (GamePad[0].active)
    {
        // Process avatar movement if the first game pad is active
    }
    else if (GameObject.gamePaused)
    {
        // Check if the player is trying to unpause the game
    }
    else if (GameObject.menuActive)
    {
```

```
        // Process input for menus
    }
    else if (GameObject.numberOfActivePlayers == 0)
    {
        // Game Over. Perform cleanup.
    }
}

public void Draw([arguments])
{
    // Determine if the avatar should be drawn
    if (GameObject.numberOfActivePlayers > 0 &&
        !GameObject.gamePaused &&
        !GameObject.gameOver)
    {
        // Draw the avatar
    }
}
```

This doesn't seem so bad for now, but what happens when the player tries to start the game from the second controller? The developer then has to go back into the code and determine why the gamepad seems to have gone dead. When the game's producer decides that the "Game Over" screen should paint the game map and all visible characters, the developer will need to update the `Draw()` method and add consideration for the "Game Over" state.

The update and draw methods start to get ugly the more advanced the application gets, because typically once the game elements are created and the game enters its processing loop, the frame-to-frame calls to methods like `Update()` and `Draw()` are usually what triggers the game elements to perform whatever processing and rendering needs to be done. As features evolve and the code becomes more complex, defensive

coding techniques start to creep into play, and large sections of functionality become wrapped in conditional blocks so that they only take effect in certain state conditions.

Using the game state observer pattern, the individual elements of the game can be immediately notified of state changes and respond to them at the time of state change as opposed to waiting for the next call to `Update()`. Game elements can also exclude themselves from rendering when the game state no longer requires or allows them to be drawn, saving valuable processing power.

The benefits of proper state management and a robust state change notification mechanism become immediately evident once you begin to use them in your own application. In the upcoming sections of this gem, we take a more in-depth look at the various software components that are required to put the game state observer pattern to work for you.

^[1]This gem also makes use of the *singleton design pattern*, but it is outside the scope of this gem. For more information regarding this and other design patterns, refer to [1].

26.1 Creating a Game State Manager

To begin using the game state observer pattern, we first need a class to represent the game state, such as the `GameState` class shown in Listing 26.1. Since a game should only ever be in one state, we'll employ another design pattern for the `GameState` implementation—the singleton design pattern.

Listing 26.1: `GameState` class implementation in C#.

```
class GameState
{
    // This enumerated type is where all valid game states are defined
    public enum State
```

```
{
    Initializing,
    StartMenu,

    Tutorial,
    InPlay,
    GameOver,
    Paused,
    BetweenLevels,
    GameEnded,
    ConfirmExit,
    GameOptionsMenu,
    DemoMode
};

// Define the one and only instance of the GameState class.
// This is the Singleton
private static GameState _instance;

// This data member will store the current state.
private State _currentState;

// This private constructor can only be called from within this
// class. This is how the Singleton pattern ensures that only
// one instance of this class will ever exist.
private GameState()
{
}

// This public accessor gives the outside world access to the
// Singleton instance of GameState.
public static GameState instance
{
```

```
get
{
    // If the instance has not been defined, create a new
    // instance.
    if (GameState._instance == null)
    {
        GameState._instance = new GameState();
    }

    // Return the instance
    return GameState._instance;
}

// These accessors allow the current state to be queried and
// set by the outside world.
public State currentState
{
    get { return this._currentState; }
    set { this._currentState = value; }
}
}
```

The inline comments in Listing 26.1 tell most of the story, but let's examine the moving parts.

- The enumerated `State` subtype (`GameState.State`) contains the "master list" of all valid game states.
- The `_instance` data member is marked as private, keeping accessibility to the data under the control of the class itself. This data member is also marked static, meaning it belongs to the class and not to the instance.
- The `_currentState` data member is also private. It stores the value of the current game state, as allowed by the `GameState.State` enumerated type.

- The only constructor provided for the `GameState` class is also marked private, meaning the class can only be instantiated from within itself.
- The `instance` accessor parameter allows the outside world to get a reference to the singleton instance of `GameState`. The retrieval mechanism first checks to see if the class has been instantiated. If not, it creates a new instance of `GameState` and stores a reference to it in the static `_instance` member. Once a valid instance has been created and stored, a reference to it is returned.
- The `currentState` accessor parameters are used to set and retrieve the current game state. While the implementations shown here are quite simple, more complex logic can be employed to ensure that all state transitions are legal.

At this point, we have a fairly simple, but functional game state manager. It has everything it needs to be a useful addition to a game or game engine project. In its current state, it can be used to establish and update the current game state, and it can be queried by other software modules that need to know the state of the game in order to function properly.

By providing other software modules with a single access point to set and retrieve the game's state, managing input processing and rendering becomes a function of the game's current state, rather than artificial conditions such as whether or not a given controller is active or has a pressed button.

The avatar `Update()` method being called in the game loop can be simplified such that the avatar only processes updates when the game is in a state that is meaningful to the avatar:

```
public void Update([arguments])
{
    if (GameState.instance.currentState != GameState.State.InPlay)
    {
        // Do nothing if the game is not in play
        return;
    }
}
```

```
}  
  
// Process this update  
}
```

Making the operations performed by each software module functions of the current game state reduces processing overhead by allowing the modules to determine if there is any need for them to perform given the current game state.

This formalized state management is helpful, but there is a great deal of room for improvement. Each game module still needs to query the game state in order to know what the current state is. Another thing to consider is that the software only has access to the current state. There is, as yet, no way for a module to be notified that the state *has changed*.

Some software modules may need to perform a set of operations when the game transitions from one state to another. For example, an automatic game save system might need to know that the player has just completed some stage of the game and has transitioned to the "stats" screen. This could be done by changing the game state from `InPlay` to `BetweenLevels` and using the game state observer pattern to notify the automatic game save mechanism to update the player's stats and inventory.

26.2 The Interfaces of the Game State Observer Pattern

In this section, we begin the process of turning the `GameState` into an "observable" object and create the foundation that will provide the communication path between the `GameState` and the other modules of your game.

In order for the software in your game engine to treat the `GameState` as the subject of observation, we need to create an interface that tells the rest of the game's software that `GameState` can be observed. The `IObservable` interface shown in Listing 26.2

provides a way to do this. As you can see, there's very little to this interface, as it only defines the two methods `Subscribe()` and `Unsubscribe()`. Some implementations of the observer pattern also define a `NotifySubscribers()` method in the `IObservable` interface, but since all interface methods must be defined as public, that would allow other software modules to force observation subjects to notify subscribers even if no change has been made. The game state observer pattern allows the `GameState` class to decide when to notify its subscribers of a change.

Listing 26.2: The `IObservable` and `INotifiable` interfaces.

```
interface IObservable
{
    void Subscribe(INotifiable observer);
    void Unsubscribe(INotifiable observer);
}

interface INotifiable
{
    void ProcessNotification(IObservable subject);
}
```

Both the `Subscribe()` and `Unsubscribe()` methods take exactly one parameter. This parameter has the type `INotifiable`, meaning it is an instance of a class that implements the `INotifiable` interface shown in Listing 26.2. The `INotifiable` interface defines only one method, `ProcessNotification()`. This method accepts one argument of type `IObservable`, meaning it is any instance of a class that implements the `IObservable` interface.

Since any class can implement an interface, you as the developer can decide which modules in your software can observe and/or be observed. To implement the interface, the class definition must first be modified to indicate that it implements a given

interface. Then it must provide an implementation for every method that the interface defines.

26.3 Making GameState Observable

To make our `GameState` class observable, we change its class declaration to read as follows: ^[2]

```
class GameState : IObservable
```

Now that we have tagged the `GameState` class as observable, we must provide the public methods necessary to satisfy the interface's requirements. But before we move on to the method implementation, let's take a brief look at how observation works in the game state observer pattern:

- The subject of observation (the `IObservable`) contains a list of subscribers.
- Any object capable of processing notifications (the `INotifiable` interface) can call the subject's `Subscribe()` method and be added to the list of subscribers.
- When the subject is modified in a way that requires notification to be sent out, the subscriber iterates over the list of observers and calls each observer's `ProcessNotification()` method.
- If an observer wants to stop receiving updates from the subject, the observer can call the subject's `Unsubscribe()` method to be removed from the list of subscribers.

The `GameState` class needs a data member in which to store its list of observers, so we add the following line of code to the `GameState` class definition under the definition of the `currentState` member:

```
private List<INotifiable> _observers = new List<INotifiable>();
```

This creates a `List` object called `_observers` that will be used to store references to `INotifiable` objects.

Now, the subscription mechanism must be created. The `Subscribe()` method is fairly simple, as shown in Listing 26.3, and can be added at the end of the `GameState` class implementation. What this method effectively does is ensure that the observer requesting notifications is not already in the observer list, and if not, then it is added to the list.

Listing 26.3: The `Subscribe()` and `Unsubscribe()` methods of the `GameState` class.

```
public void Subscribe(INotifiable observer)
{
    if (!this._observers.Contains(observer))
    {
        this._observers.Add(observer);
    }
}

public void Unsubscribe(INotifiable observer)
{
    if (this._observers.Contains(observer))
    {
        this._observers.Remove(observer);
    }
}
```

With the subscription method in place, we need a method to unsubscribe as well. The `Unsubscribe()` method can be added beneath the `Subscribe()` method in the `GameState` class with the implementation shown in Listing 26.3. In the same fashion as the `Subscribe()` method, the `Unsubscribe()` method checks to see if the observer requesting removal is in the list, and the observer is only removed from the list if it is found.

At this point, we have an observable `GameState` class. Subscribers throughout the

game application can register with the `GameState` instance to be informed of modifications to the state, but there's still some work left to do.

The `GameState` class we have so far does not yet notify its subscribers. It simply manages a list of interested software components. We still need to provide the `GameState` class with a method that notifies its observers of state changes. The `_NotifySubscribers()` method shown in Listing 26.4 can be added to the bottom of the `GameState` class implementation to take care of this. The `_NotifySubscribers()` method iterates over the list of observers, and for each one calls its `ProcessNotification()` method with a reference to the singleton `GameState` instance (`this`).

Listing 26.4: The `_NotifySubscribers()` method of the `GameState` class.

```
private void _NotifySubscribers()
{
    foreach (INotifiable observer in this._observers)
    {
        observer.ProcessNotification(this);
    }
}
```

The `GameState` class now has a method for notifying its observers, but the method isn't being called anywhere. The next step is to update the state modification accessor method to call `_NotifySubscribers()` when the game state is changed, as shown in Listing 26.5. The new accessor to set the current state only does something when state is actually changing, and it calls `_NotifySubscribers()` to send all the observers the update.

Listing 26.5: The set implementation for the `currentState` member of the `GameState` class.

```
public State currentState
{
    get { return this._currentState; }

    set
    {
        if (this._currentState != value)
        {
            this._currentState = value;
            this._NotifySubscribers();
        }
    }
}
```

That's about all there is for the game state manager's role in its own observation. The ball is now in the observer's court. Having received an update, it must be able to take action based on the data it has received.

^[2]This is a C# implementation, and a C++ implementation is similar. Other languages provide keywords such as `implements` or `extends` to indicate relationships among various structures. Be sure to use the appropriate syntax for the programming language your game uses.

26.4 Creating Observers

Any class in your game engine library can be made into an observer, and thus can be made to observe game state. To turn an existing class into an observer, you must first declare that it implements the `INotifiable` interface, and then you must provide the implementation for the `ProcessNotification()` method defined in the `INotifiable` interface, as exemplified by Listing 26.6. In its minimalist form, `SomeGameComponent` is an observer. It defines no useful functionality, stores no data, and does nothing with

any notifications it receives, but it has all the moving parts needed to be classified as an observer.

Listing 26.6: A sample observer implementation.

```
class SomeGameComponent : INotifiable
{
    // Define class data members here

    public void ProcessNotification(IObservable subject)
    {
        // Query the subject for any meaningful changes
    }
}
```

In Listing 26.6, the argument received by the `ProcessNotification()` method is of type `IObservable`. This means that any instance of any class that implements the `IObservable` interface can be passed into this method. However, it also means that only the methods defined in the `IObservable` interface can legally be called without explicitly casting the argument to a known type. In other words, we cannot query `GameState` properties or call methods of the `GameState` class without casting `subject` to be of type `GameState`, as shown in Listing 26.7.

Listing 26.7: Accessing an observer by its native type. ^[3]

```
public void ProcessNotification(IObservable subject)
{
    // Cast the instance to its native type
    GameState gsSubject = (GameState) subject;

    // Use the qualified reference to access its data and methods
    GameState.State currentState = gsSubject.currentState;
```



```
}
```

The implementation for the `ProcessNotification()` method is functional, but somewhat limiting. Suppose you have a game component in your library that needs to observe multiple subjects of various types. The above implementation would fail if subject is not of type `GameState`. To manage this situation, we can provide a switchboard mechanism within `ProcessNotification()` that doesn't handle the heavy lifting, but simply identifies the best method for the job.

Listing 26.8 shows how a switchboard mechanism could be used to funnel notification processing through type-specific methods so that any class that implements `INotifiable` can observe multiple subjects, regardless of the subject's native type. When the type is identified, processing is handed off to a purpose-built method capable of handling update notifications for that type of class. `SomeObservable`, in the example above, can be thought of as some other class that implements the `IObservable` interface. The `ProcessNotification()` method could get a bit unwieldy, but in practical cases, it's uncommon to observe subjects of more than a handful of types.

Listing 26.8: Supporting multiple subject types.

```
public void ProcessNotification(IObservable subject)
{
    // Determine the best method to handle this update
    if (subject.GetType() == typeof(GameState))
    {
        this._ProcessNotification((GameState) subject);
    }
    else if (subject.GetType() == typeof(SomeObservable))
    {
        this._ProcessNotification((SomeObservable) subject);
    }
}
```

```
    }  
}  
  
protected void _ProcessNotification(GameState subject)  
{  
    // Process notifications for instances of GameState  
}  
  
protected void _ProcessNotification(SomeObservable subject)  
{  
    // Process notifications for instances of SomeObservable  
}
```

^[3]Listing 26.7 shows how to cast an observable subject to its native type so that its data members and methods can be accessed. Bear in mind that the syntax for reference casting may be different in the programming language you are using. In C++, you would normally use `static_cast` to perform the cast to the derived class type.

26.5 Managing Functionality by Game State

In a real game application, we'd want our objects and game entities to react in meaningful ways when a notification of change has been received. When working with game state this typically means the game needs to react differently to controller input or render a different scene or menu. Imagine a game engine with a series of queues that provide different functionality. For example, modules that want to process controller input could be placed in the "input queue", and models or sprites that need to be drawn could be placed in the "render queue". These queues can provide methods to allow objects to be added and removed from them.

Upon receipt of notification of a state change, a game module can add or remove

itself from the above queues proactively. The avatar management code can stop processing input when the game is in a paused state, and the menu management software can stop rendering menus when the game is in play.

A full example of such a mechanism is well beyond the scope of this gem, but a simple example can be found on the accompanying CD. The example, called `observerSample`, is a small XNA project that shows how the modification of game state can be communicated to all the interested modules of a game application to manage which components process input and render to the screen.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

27

Fast Trigonometric Operations Using Cordic Methods

John Bolton

Netflix

Overview

Trigonometric functions are required to display 3D and rotating 2D graphics, but these functions are generally not well-supported on some game platforms such as handheld devices and cellular phones. On platforms without a floating-point processor, trigonometric functions may instead be implemented by emulating floating-point number representations and operations, and performance can be very poor as a result. In contrast, CORDIC methods implement standard trigonometric functions using simple integer math and bit shifting, and this can be extremely fast.

CORDIC methods were invented by Jack Volder [2] in the late 1950s as a way to compute trigonometric functions in hardware for use in avionics. CORDIC stands for COordinate Rotation DIgital Computer. Later, the methods were extended by John Walther [3] and others to related functions (hyperbolic and exponential functions, for example).

27.1 Rotation Mode Algorithm

The CORDIC methods are based on iteratively rotating a point by fixed angles until a desired rotation is achieved. The equations for rotating a point (x,y) about the origin in two dimensions are

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta,$$

and these are equivalent to

$$x' = \cos\theta(x - y\tan\theta)$$

$$y' = \cos\theta(y + x\tan\theta).$$

The main concept behind the CORDIC methods is to iteratively rotate the point by angles whose tangent is a power of two until the desired angle is reached using the formula

$$x_{i+1} = C_i(x_i - d_i y_i \cdot 2^{-i})$$

$$y_{i+1} = C_i(y_i + d_i x_i \cdot 2^{-i}),$$

where

$$C_i = \cos(\tan^{-1} 2^{-i}) = \frac{1}{\sqrt{1 + 2^{-2i}}}$$

Choosing a power of two allows the multiplication by $\tan\theta$ to be replaced with a shift operation.

Because the values of C_i are constant, the multiplication by C_i in each iteration can be moved out of the iteration process, and the result can be adjusted by doing a single accumulated multiplication instead (when necessary). This optimization step reduces each iteration to a few arithmetic and shift operations plus an indexed look-up into an array of angle values.

$$C = C_0 C_1 \cdots C_{n-1} = \prod_0^{n-1} \frac{1}{\sqrt{1 + 2^{-2i}}}$$

The value of C is approximately 0.60725 as the number of iterations approaches infinity, though the actual value is based on a finite number of iterations.

In each iteration, the point is rotated by successively smaller amounts. A third iterated value a_i holds the difference between the accumulated rotation angle and the desired rotation angle, and is used to determine if the next iteration should rotate the point clockwise or counterclockwise. a_0 is initialized to the input angle, and a_i approaches zero. The direction of rotation is determined by the sign of the difference and is represented here by the value d_i :

$$a_0 = \theta$$

$$a_{i+1} = a_i - d_i \tan^{-1} 2^{-i}$$

$$d_i = \begin{cases} -1, & \text{if } a_i < 0 ; \\ +1, & \text{if } a_i \geq 0 . \end{cases}$$

As mentioned earlier, the point is rotated by successively smaller amounts. The iteration continues until the amount of rotation is too small to be represented by the chosen fixed-point format. Using fewer iterations is possible, but produces less accurate results. The number of iterations is 25 when using an 8.24-bit fixed-point format and measuring angles in radians.

This basic algorithm is called the "rotation" mode and can be used to rotate an arbitrary 2D vector. It can also compute the sine and cosine of an angle simply by rotating the point (1,0) by that angle and returning the resulting values. It is important to note that this algorithm requires the input angle to be in the range $[-\pi/2, \pi/2]$. For angles outside of this range, the point is first rotated by 2π and π as necessary until the angle is in the proper range.

In summary, the values of the last iteration of the rotation mode algorithm are given by

$$x_n = x_0 \cos \theta - y_0 \sin \theta$$

$$y_n = x_0 \sin \theta + y_0 \cos \theta$$

$$\alpha_n = \theta$$

27.2 Vectoring Mode Algorithm

A second related algorithm iteratively rotates a given point towards the x -axis. This is called the "vectoring" mode. In this algorithm, the point is rotated by successively smaller amounts until the value of the y -coordinate is zero. As in the rotation mode, the direction of rotation is represented by the value d_i , but this is instead determined by the sign of the y -coordinate, rather than the angle:

$$d_i = \begin{cases} -1, & \text{if } y_i \geq 0 ; \\ +1, & \text{if } y_i < 0 . \end{cases}$$

Again, it is important to note that this algorithm requires the angle of the input vector to be in the range $[-\pi/2, \pi/2]$. For angles outside of this range, the point is first rotated by 2π and π as necessary until it is in the proper range.

After completion, the accumulated angle is the angle between the initial point and the x -axis, and the value of the x -coordinate is the distance to the point from the origin.

In summary, the values of the last iteration of the vectoring mode algorithm are given by

$$x_n = \sqrt{x_0^2 + y_0^2}$$

$$y_n = 0$$

$$a_n = \tan^{-1}\left(\frac{y_0}{x_0}\right).$$

27.3 Applications

The following table summarizes the computations that can be done by these two algorithms, given an initial vector (x_0, y_0) and an angle θ :

Operation	Mode	Input	Results
Sine/cosine	rotation	$(1,0), \theta$	$\cos\theta = x_n$ $\sin\theta = y_n$
Arctangent	vectoring	$(x_0, y_0), \theta$	$\tan^{-1}(y_0/x_0) + \theta = \alpha_n$
2D vector rotation	rotation	$(x_0, y_0), \theta$	$(x', y') = (x_n, y_n)$
Vector length	vectoring	(x_0, y_0)	$\ (x_0, y_0)\ = x_n$

27.4 Implementation

The following code listings show implementations of all the algorithms and functions listed above. In these implementations, the fixed-point format is assumed to use an 8-bit whole part and 24-bit fraction part.

The code in Listing 27.1 implements the two algorithms. Note that the

multiplication by C is not present here and must be handled elsewhere. In some cases, multiplication by C is not necessary because either the scale of the result is irrelevant or because there is a more efficient way to apply the value. Refer to each application in the listings that follow to see how the value of C is applied.

Listing 27.1: Rotation and vectoring mode implementations.

```
// Returns 0 if n >= 0, and -1 if n < 0
inline int32 S(int32 n)
{
    return n >> (sizeof(int32) * 8 - 1);
}

// Returns n if d == 0, and -n if d == -1
inline int32 CONDITIONAL_NEG(int32 n, int32 d)
{
    return (n ^ d) - d;
}

void RotationMode(int32 x, int32 y, int32 a, int32 *rx, int32 *ry)
{
    for (int i = 0; i < NUMBER_OF_ITERATIONS; ++i)
    {
        int32 d = S(a);    // (a >= 0) ? 0 : -1;
        int32 xi = x;
        int32 yi = y;
        x = x - CONDITIONAL_NEG(yi >> i, d);
        y = y + CONDITIONAL_NEG(xi >> i, d);
        a -= CONDITIONAL_NEG(angles[i], d);
    }

    *rx = x;
```

```

    *ry = y;
}

void VectoringMode(int32 x, int32 y, int32 a, int32 *rl, int32 *ra)
{
    for (int i = 0; i < NUMBER_OF_ITERATIONS; ++i)
    {
        int32 d = S(y);    // (y >= 0) ? 0 : -1;
        int32 xi = x;
        int32 yi = y;
        x = x + CONDITIONAL_NEG(yi >> i, d);
        y = y - CONDITIONAL_NEG(xi >> i, d);
        a += CONDITIONAL_NEG(angles[i], d);
    }

    *rl = x;
    *ra = a;
}

```

In rotation mode, the input angle must be in the range $[-\pi/2, \pi/2]$. The function shown in Listing 27.2 rotates the input vector by multiples of 2π and π , adjusting the input angle accordingly.

Listing 27.2: Normalizing the input range.

```

void Normalize(int32& x, int32& y, int32& a)
{
    while (a >= FIXED_TWO_PI) a -= FIXED_TWO_PI;
    while (a <= -FIXED_TWO_PI) a += FIXED_TWO_PI;

    while (a > FIXED_PI_OVER_2)
    {
        x = -x;

```

```

    y = -y;
    a -= FIXED_PI;
}

while (a < -FIXED_PI_OVER_2)
{
    x = -x;
    y = -y;
    a += FIXED_PI;
}
}

```

The table of angles is built by computing the values of $\tan^{-1} 2^{-i}$ in the appropriate fixed-point format until the value is zero. The number of iterations in the rotation and vectoring mode algorithms is simply the number of entries in the table. This table can be precomputed as it is not likely to vary. The code in Listing 27.3 shows how this is done.

Listing 27.3: Angle table generation.

```

vector<int32> angles;
int i = 0;
for (;;)
{
    double a = atan(pow(2.0, -i));
    int32 fixed_a = int32(a * 0x01000000 + 0.5);
    if (fixed_a <= 0) break;

    angles.push_back(fixed_a);
    ++i;
}
int NUMBER_OF_ITERATIONS = angles.size();

```

The code in Listing 27.4 shows how the value of C is computed. This value can be precomputed, as it is not likely to vary.

Listing 27.4: Computation of C .

```
double k = 1.0;
for (int i = 0; i < NUMBER_OF_ITERATIONS; ++i)
{
    k *= sqrt(1.0 + pow(4.0, -i));
}

int32 C = int32(1.0 / k * 0x01000000 + 0.5);
```

Sine and Cosine

The code in Listing 27.5 uses the rotation mode to compute the sine and cosine of an angle.

Listing 27.5: Sine and cosine implementation.

```
void SineCosine(int32 a, int32& s, int32& c)
{
    c = C;    // Pre-multiply (1, 0) by C
    s = 0;
    Normalize(c, s, a);    // Adjust angle to the range [-pi/2, pi/2]
    RotationMode(c, s, a, &c, &s);
}
```

Arctangent

The code in Listing 27.6 uses the vectoring mode to compute the arctangent of a value.

Listing 27.6: Arctangent implementation.

```
int32 ArcTangent(int32 m)
{
    int 32    angle, length;

    VectoringMode(0x01000000, m, 0, &length, &angle);
    return a;
}
```

2D Vector Rotation

The code in Listing 27.7 uses the rotation mode to rotate a vector by a given angle.

Listing 27.7: Vector rotation implementation.

```
void Rotate(int32& x, int32& y, int32 a)
{
    Normalize(x, y, a);    // Adjust angle to the range [-pi/2, pi/2]
    RotationMode(x, y, a, &x, &y);

    // The vector must be scaled by C
    x = int32((int64(x) * int64(C)) >> 24);
    y = int32((int64(y) * int64(C)) >> 24);
}
```

Vector Length

The code in Listing 27.8 uses the vectoring mode to compute the length of a vector.

Listing 27.8: Vector length implementation.

```
int32 Length(int32 x, int32 y)
{
    int32    angle, length;
```

```
x = abs(x); // Put the vector into the range [-pi/2, pi/2]
VectoringMode(x, y, 0, &length, &angle);

// The length must be scaled by C
return int32((int64(length) * int64(C)) >> 24);
}
```

27.5 Considerations

When using a fixed-point format, overflow and precision are a constant concern. The following considerations must be taken into account when using these functions:

1. The length of the vector(x_i, y_i) will grow as it is rotated by a factor of approximately 1.65. You must constrain the input values to ensure that this will not cause an overflow.
2. Certain optimizations in the code presented here are done in order to eliminate branching and multiplication. Optimizations such as these can be tailored to the target platform.
3. The code implemented here assumes that the compiler implements the shift operator on signed types using an arithmetic shift. In C/C++, the precise behavior of the shift operator on signed integer types is defined by the compiler implementation. The compiler may or may not use an arithmetic shift in this case. For example, the result of shifting the value -1 right by one bit may be 0x7FFFFFFF or 0xFFFFFFFF, depending on the compiler.

27.6 Extensions

The hyperbolic equivalents, as well as the inverses of the functions presented above, can also be computed using similar methods. In addition, functions such as tangent, hyperbolic tangent, e^x , natural log, and square root can be derived from the basic

functions. Andraka [1] and Walther [3] describe the implementation of these extensions.

References

- [1] Ray Andraka. "A survey of CORDIC algorithms for FPGA based computers". *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, 1998, pp. 191–200.
- [2] Jack E. Volder. "The CORDIC Trigonometric Computing Technique". *IRE Transactions on Electronic Computing*, Volume EC-8 (September 1959), pp. 330–334.
- [3] John S. Walther. "A Unified Algorithm for Elementary Functions". *Spring Joint Computer Conference Proceedings*, Volume 38 (1971), pp. 379–385.

28

Inter-Process Communication Based on Your Own RPC Subsystem

Kurt Pelzer

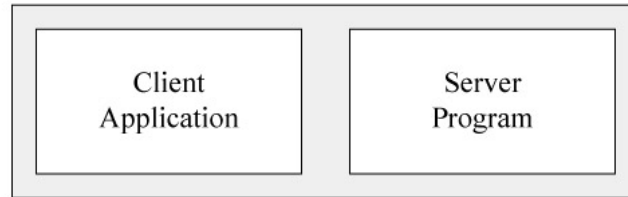
Piranha Bytes

Overview

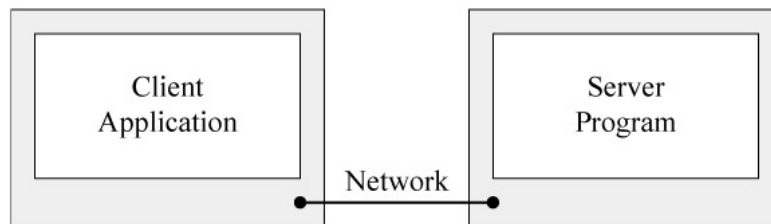
The remote procedure call (RPC) technique is a powerful tool for constructing distributed applications. It implements a client/server based system without requiring that callers be aware of the underlying network. That is, the programmer would write essentially the same code whether the procedure is local to the executing program or remote. RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports (e.g., TCP/IP or UDP/IP).

When an application is combined with an RPC subsystem, it is able to interact with a second application (e.g., editor and game), and it transparently makes remote calls through a local procedure interface. The two processes may be on the same system as in Figure 28.1(a), or they may be on different systems with a network connecting them as in Figure 28.1(b).

(a) Client and Server Application on the same machine



(b) Client and Server Application on different machines



Same in cases (a) and (b):

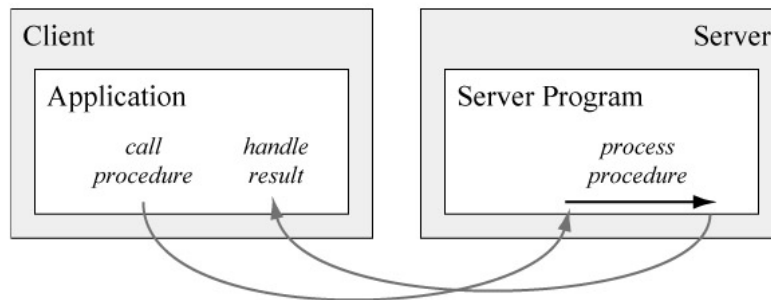


Figure 28.1: Two applications connected via RPC—Client and server applications on (a) the same or (b) different machines.

Because of its transport independence, RPC makes the client/server model of computing more powerful and easier to program. It is based on extending the notion of conventional, or local, procedure calling so that the called procedure need not exist in the same address space as the calling procedure. Implementing your own RPC system is useful because it enables you to connect different systems like PCs and multimedia

game consoles such as the PlayStation 3 or Xbox 360. It is easy to use RPC in your own application, so it makes sense to integrate it into your engine and development tools.

28.1 History of Remote Procedure Call

The idea of the remote procedure call goes back to 1976, when it was described in RFC 707 [1] as an inter-process communication (IPC) technology. An IPC is a set of techniques for the exchange of data among multiple threads in one or more processes that may be running on one or more computers connected by a network. IPC techniques are divided into methods for message passing, synchronization, shared memory, and remote procedure calls. One of the first business uses of RPC was by Xerox under the name "Courier" in 1981 [2].

The first popular implementation of RPC was Sun's RPC, now called ONC RPC. It is still widely used today on several platforms [3, 4]. Another early implementation was Apollo Computer's NCS (Network Computing System). It was used as the foundation of DCE/RPC. A decade later (in the mid 1990s), Microsoft adopted DCE/RPC as the basis of Microsoft RPC (MSRPC), and implemented DCOM atop it.

28.2 How RPC Works: Internal Architecture of RPC

An RPC is initiated by the client sending a request message to a known remote server in order to execute a specified procedure using supplied parameters. A response is returned to the client where the application continues along with its process. While the server is processing the call, the client is blocked—it waits until the server has finished processing before resuming execution.

Figure 28.2 shows the flow of activity that takes place during an RPC call between two networked systems. Like a function call, when an RPC is made, the calling

arguments are passed to the remote procedure, and the caller waits for a response to be returned from the remote procedure. The steps are the following:

1. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received or the RPC times out.
2. When the request arrives, the server calls a dispatch routine that prepares the requested service.
3. The requested service is performed on the server.
4. The server sends the result to the client.
5. After the RPC is completed, the client program continues.

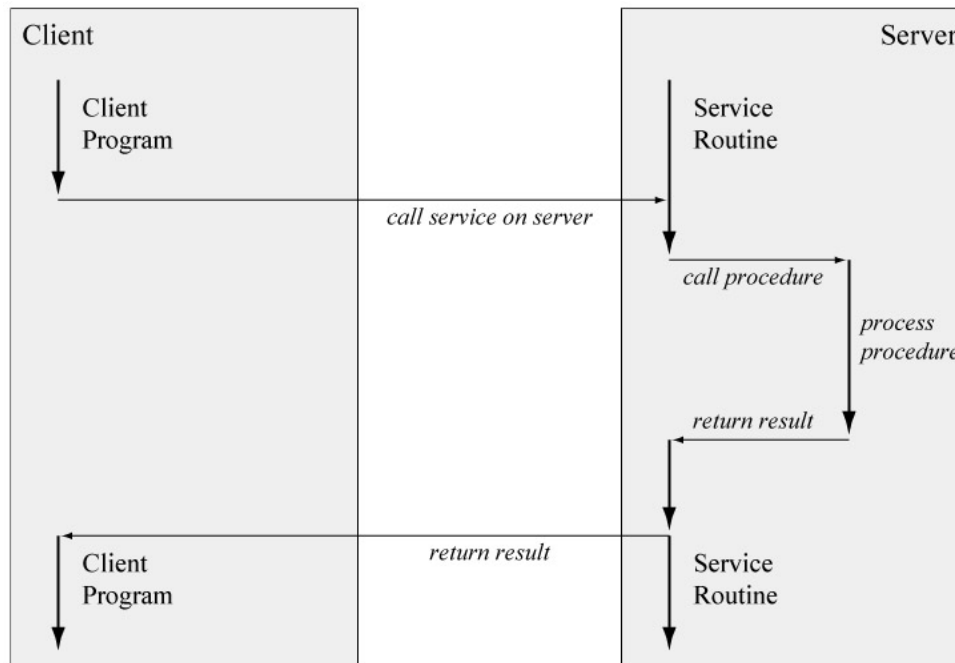


Figure 28.2: Steps during an RPC call, initiated by the client sending a request to the server.

An important difference between remote procedure calls and local calls is that remote calls can fail because of unpredictable network problems. Also, callers generally must deal with such failures without knowing whether the remote procedure was actually invoked.

Note that in this remote procedure call model, only one of the two processes is active at any given time. However, this scenario is given only as an example. The RPC protocol makes no restrictions on concurrency, and other scenarios are possible. For example, an implementation may choose to have asynchronous RPC calls so the client may do useful work while waiting for the reply from the server. Another possibility is to have the server create a separate task to process an incoming request so the server can be free to receive other requests.

Code that calls remotely makes use of a low-level subsystem. The encoding and decoding of procedure calls is handled in a special stub module (see Figure 28.3). That RPC stub module handles the procedure identification and the marshalling of the supplied procedure parameters inside a message that has to be sent or has been received. The RPC protocol is independent of transport protocols; that is, RPC does not care how a message is passed from one process to another — the protocol is concerned only with the specification and interpretation of messages. For example, RPC may be implemented on top of TCP/IP or UDP/IP. Also, the act of binding a client to a server is not part of the RPC subsystem. This function is left to some higher-level software module.

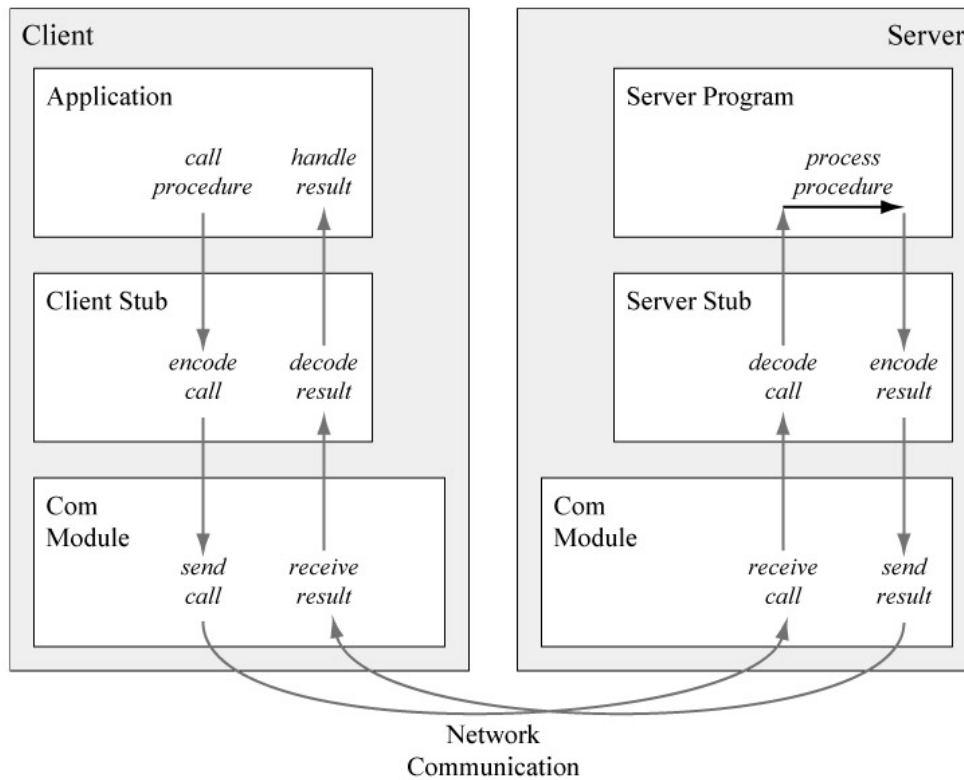


Figure 28.3: Encoding and decoding of procedure calls in special stub modules.

A remote procedure is uniquely identified by the following three pieces of information:

- A. The program number.
- B. The version number.
- C. The procedure number.

The program number A identifies a group of related remote procedures, each of which has a unique procedure number. A program may consist of one or more versions, and each version consists of a collection of procedures that are available to be called remotely. The version number B enables multiple versions of an RPC protocol to be

available simultaneously. Each version contains a number of procedures that can be called remotely, and each procedure has a procedure number C.

28.3 How to Build Your Own RPC Subsystem

We have seen that the RPC technology must allow a computer program to cause a procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That means that the RPC stub modules have to handle a number of tasks. On the client side, the stub has to hide the fact that the called procedure is going to run in a different process (on the same machine or on a different machine). On the server side, the stub has to hide that the procedure call was initiated in a different (client) process and that the result is going to be sent back. These tasks the stubs have to handle lead to a list of points that should be kept in mind when you start to implement your own RPC subsystem.

The remote procedure call mechanism must behave similarly to that of the local procedure call model. With the local model, the caller places arguments to a procedure in a well-specified location (such as in particular registers or on the stack) and transfers control to the procedure. When the caller eventually regains control, it extracts the result of the procedure from the well-specified location and continues execution.

With respect to the remote procedure call paradigm, a client calls the stub version of the wanted procedure to initiate the processing of the wanted calculations. Now, the called function in the client RPC stub module has to encode all needed information in a data packet to be able to force the remote processing of the wanted procedure by sending this packet to a server. This means that the procedure identification and all function parameters have to be encoded in this data packet. On the server side, a process is waiting for the arrival of a client message. When a call message arrives, the server runs

a dispatch routine in its RPC stub that extracts the procedure identification and its parameters, performs the requested procedure to compute the results, encodes these results in a new data packet, and sends it back to the client. Then the server waits for the arrival of the next call message. The resulting packet of the procedure call returns to the client where it has to be dispatched. Finally, the procedure that initiated the processing of the wanted calculations regains control, continues execution, and can handle the result (see Figure 28.4).

We have already seen that the identification of the wanted remote procedure must be encoded in the client RPC stub in a set of three numbers: the program number A, the version number B, and the procedure number C. That information enables the receiving and decoding RPC stub module in the server application to identify the function that must be processed. Beside this procedure identification, the client/server stub modules have to handle the marshalling of the procedure parameters. There are three different solutions for passing the parameters:

- A parameter can be passed by value—this means a local value that can be modified.
- A parameter can be passed by reference—this means the parameter is a pointer to a value that must be handled via call-to-copy/restore.
- A parameter can be a pointer to a complex data structure such as a list, tree, etc. The server could read structure elements from the client one at a time, but this would be very inefficient. A better way is to copy the complete data structure to the server address space.

With respect to possible different representation of integer and float values, characters, and other data on client and server machines, you have to be able to handle system-specific issues, especially the mixing of little endian and big endian byte ordering. You have to encode information about the data format of the packed parameters, or you have to use a machine-independent transfer format for data.

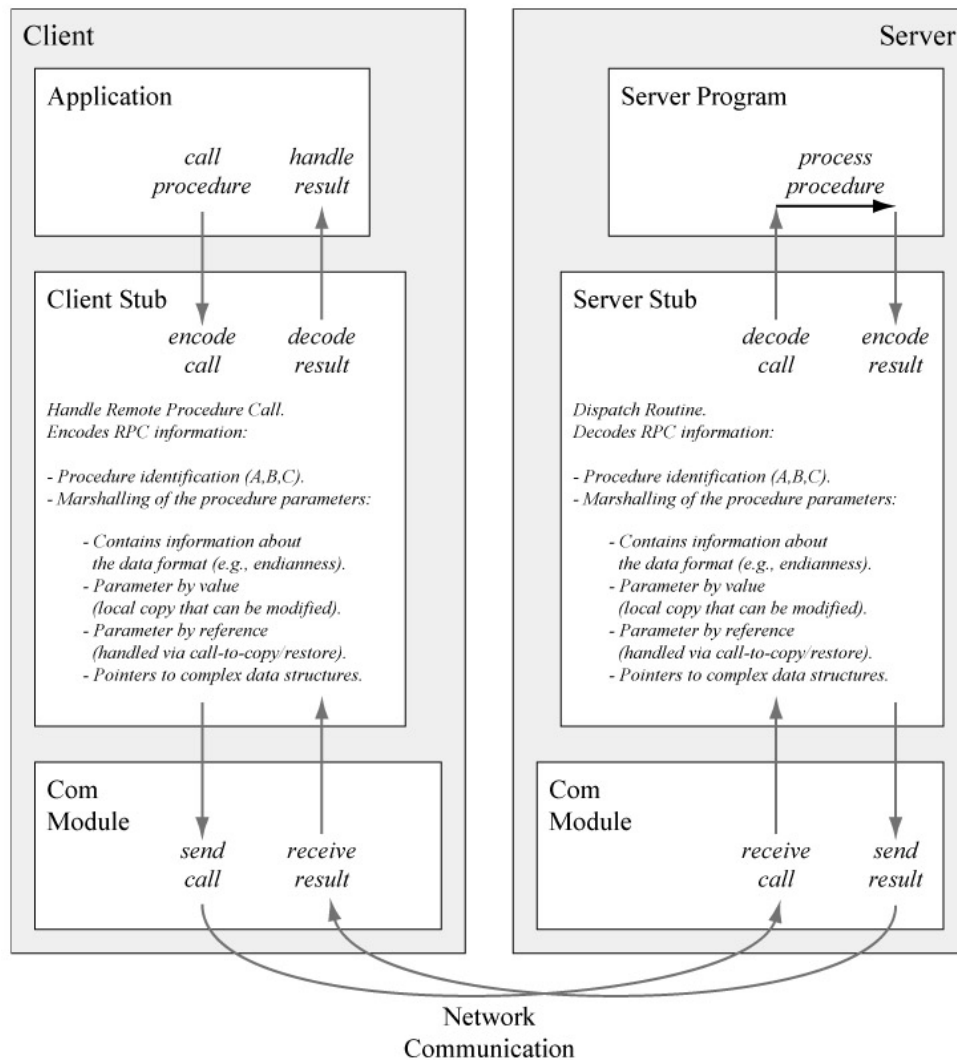


Figure 28.4: RPC stub modules handling the procedure identification and the marshalling of the supplied procedure parameters.

The simple code fragments shown in Listing 28.1 should give you an idea of how to implement the stub functions for client and server machines.

Listing 28.1: Example stub functions for the client and server.

```
// IN THE CLIENT RPC STUB

// example of a client stub version of "procedureA"
// client application calls this local procedure to run
// "procedureA" in the server application

int ClientRPCStub::procedureA(int parameter)
{
    // encode a data packet
    Pack outPacket(getProcID(this), getMshParams(parameter));

    // send packet to server
    getComModul().send(outPacket);

    // wait for server response
    ResultPack resultPacket = waitForResult();

    // dispatch the result and return to caller
    return resultPacket.getReturnValue();
}

// IN THE SERVER RPC STUB

// function to handle client requests in the server app
void ServerRPCStub::handle(Pack& inPacket)
{
    // dispatch a data packet
    ProcID procID(inPacket);
    ParamBlock params(inPacket);

    // call dispatch function to process the wanted procedure
```

```
ResultPack resultPacket = run(procID, params);

// send result to caller
getComModul().send(resultPacket);
}

// dispatch function in the server application
Pack ServerRPCStub::run(ProcID& procID, Block& params)
{
    // detect the wanted procedure and extract the parameters
    ...

    // call the local version of wanted "procedureA"

    int result = procedureA(parameter);

    // encode the packet that has to be returned to client
    return ResultPack(procID, result);
}
```

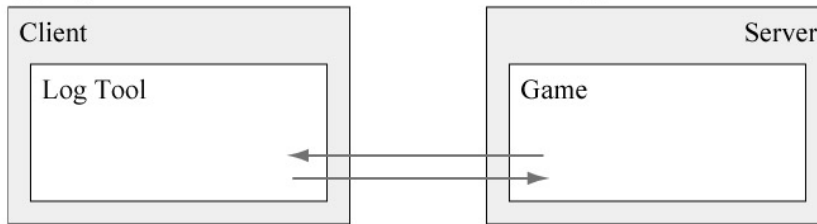
28.4 Why RPC is Useful for Game Engines

Since the RPC technology enables you to connect running processes on different systems like PC and game consoles (e.g., Xbox 360 or PlayStation 3), the possibility of distributed processing leads to many useful cases that let your engine and development tools interact at runtime. The following are some useful cases of interacting applications:

- An advanced log tool, as shown in Figure 28.5(a). This tool could be able to output the text messages (events, warnings, errors) for the sending game and additionally support an integrated console to send back commands from the tool to the game. Enabling the tool to cheat or to switch modes in the running game from outside, this is a useful application of RPC especially on game consoles that don't support

keyboards. This is also useful in the case of a long distance between the system running the log tool and the system running the game connected via TCP/IP and the local intranet or the internet.

(a) Log tool connected to an instance of the running game.



(b) Editor connected to two instances of the running game on two different platforms.

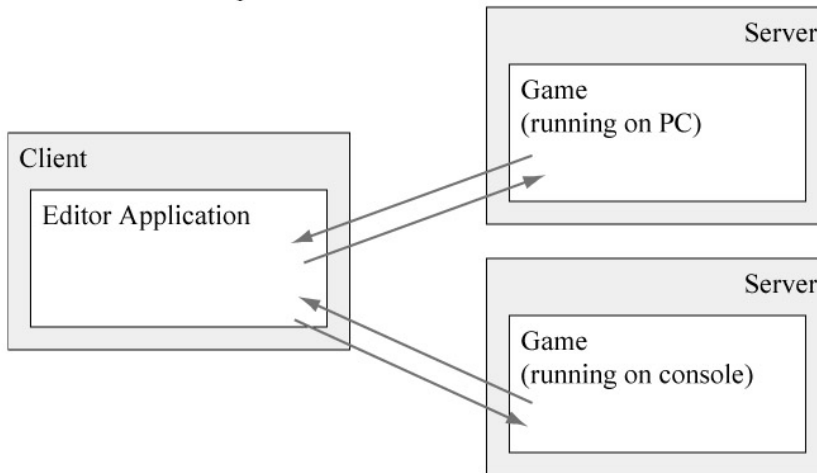


Figure 28.5: (a) A log tool connected to a running game. (b) An editor application connected to two instances of a game.

- External editing tools connected to separate runtime applications—applications that make use of connected runtime processes to be able to send instructions and receive real-time feedback. This is useful for light source placement or cutscene editing in the running game from outside with a connected tool. It is also useful for editors that

need to run system-dependent processes and calculations on other machines to be able to detect specific restrictions and issues.

- An editor connected to several platforms at the same time (see Figure 28.5(b)). It can be useful to be able to get rendered screen views from different machines at the same time. That enables you to display the screens of multiple instances of a running game sent from connected PCs and multimedia consoles on your PC and detect machine-dependent differences in quality and performance.
- Advanced precalculations on multiple machines. You can use the computer systems connected to your intranet for distributed computations in expensive precomputation steps, for example, to calculate the static lighting and ambient occlusion in high-detailed game worlds, etc.

References

- [1] James E. White. "A High-Level Framework for Network-Based Resource Sharing". Augmentation Research Center, Stanford Research Institute.
<http://tools.ietf.org/html/rfc707>
- [2] Andrew D. Birrell and Bruce Jay Nelson. "Implementing Remote Procedure Calls". Xerox Palo Alto Research Center, 1984.
<http://www.cs.yale.edu/homes/arvind/cs422/doc/rpc.pdf>
- [3] "Remote Procedure Call—OMC RPC Protocol Specification Version 1". Sun Microsystems, 1988. <http://tools.ietf.org/html/rfc1057>
- [4] "Remote Procedure Call—OMC RPC Protocol Specification Version 2". Sun Microsystems, 2009. <http://tools.ietf.org/html/rfc5531>

