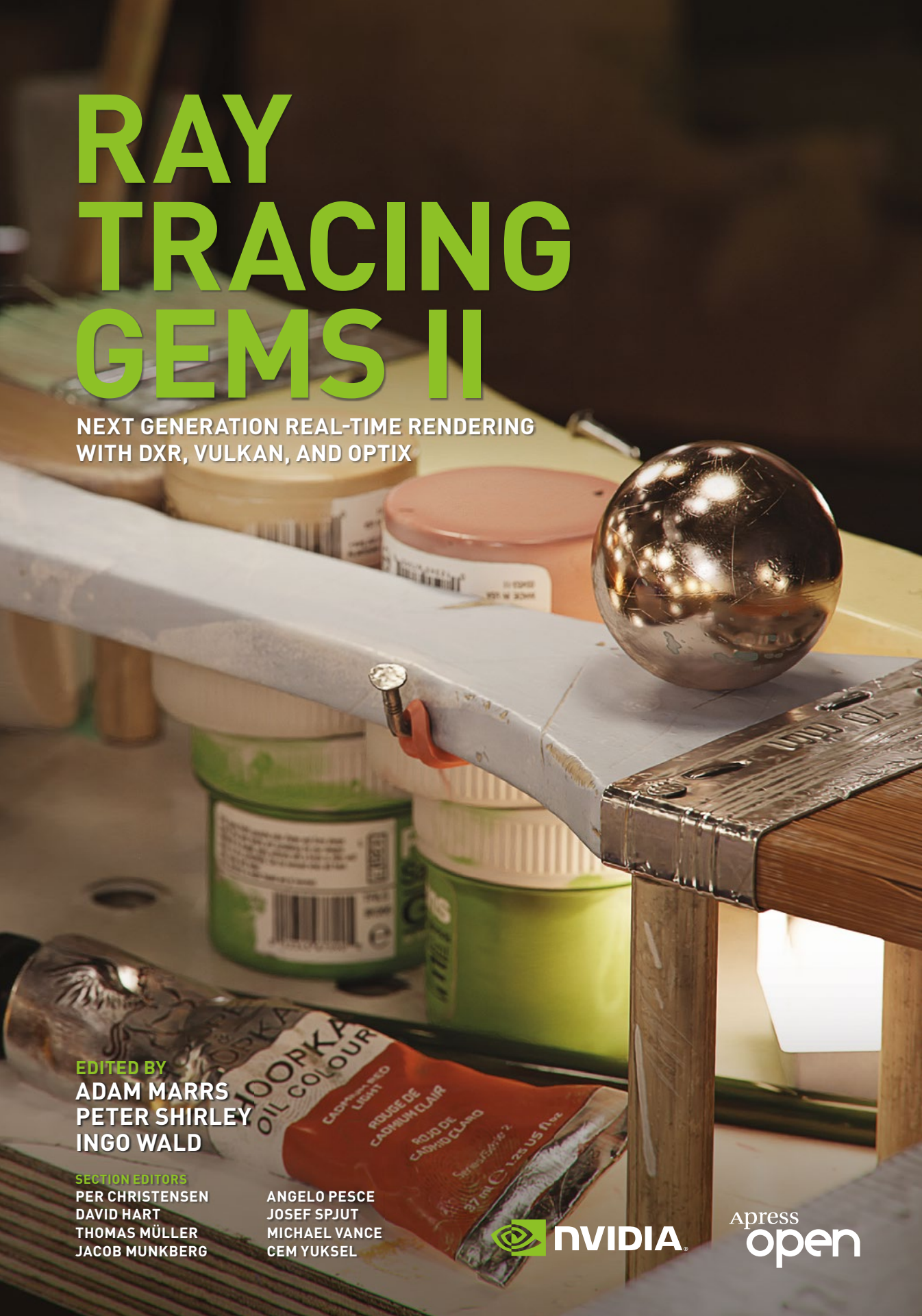

RAY
TRACING
GEMS II
NEXT GENERATION REAL-TIME RENDERING
WITH DXR, VULKAN, AND OPTIX
EDITED BY
ADAM MARRS
PETER SHIRLEY
INGO WALD
SECTION EDITORS
PER CHRISTENSEN
DAVID HART
THOMAS MÜLLER
JACOB MUNKBERG
ANGELO PESCE
JOSEF SPJUT
MICHAEL VANCE
CEM YUKSEL
NVIDIA
Apress
open

Ray Tracing Gems II

Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX

Edited by
Adam Marrs, Peter Shirley, and Ingo Wald

Section Editors
Per Christensen
David Hart
Thomas Müller
Jacob Munkberg
Angelo Pesce
Josef Spjut
Michael Vance
Cem Yuksel





Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX

Edited by
Adam Marrs
Peter Shirley
Ingo Wald

Section Editors
Per Christensen
David Hart
Thomas Müller
Jacob Munkberg
Angelo Pesce
Josef Spjut
Michael Vance
Cem Yuksel

ISBN-13 (pbk): 978-1-4842-7184-1
ISBN-13 (electronic): 978-1-4842-7185-8
https://doi.org/10.1007/978-1-4842-7185-8

Copyright © 2021 by NVIDIA

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark. The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Open Access This book is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this book or parts of it.

The images or other third party material in this book are included in the book's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the book's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editors: Susan McDermott, Natalie Pao
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Cover image designed by NVIDIA

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484271841. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

Preface ..... **xix**

Foreword ..... **xxiii**

Contributors ..... **xxix**

Notation ..... **lv**

PART I: Ray Tracing Foundations **5**

Chapter 1: A Breakneck Summary of Photographic Terms (and Their Utility to Ray Tracing) **9**

1.1 Introduction 9

1.2 Digital Sensor Technology 10

1.3 Film 13

1.4 Common Capture Dimensions 14

1.5 Common Capture Resolutions 16

1.6 Lensing 16

1.7 Shutter 19

1.8 Exposure 21

1.9 Equivalency 24

1.10 Physical Lenses 26

1.11 Bokeh 27

1.12 Various Lens Imperfections 29

1.13 Optical Elements 33

1.14 Anamorphic 34

1.15 Camera Movement 34

Chapter 2: Ray Axis-Aligned Bounding Box Intersection 37

2.1 The Method 37

Chapter 3: Essential Ray Generation Shaders 41

3.1 Introduction 41

3.2 Camera Rays 42

3.3 Pinhole Perspective 47

3.4 Thin Lens 50

3.5 Generalized Panini 52

3.6 Fisheye 55

3.7 Lenslet 56

3.8 Octahedral 58

3.9 Cube Map 60

3.10 Orthographic 61

3.11 Fibonacci Sphere 62

Chapter 4: Hacking the Shadow Terminator 65

4.1 Introduction 66

4.2 Related Work 68

4.3 Moving the Intersection Point in Hindsight 70

4.4 Analysis 72

4.5 Discussion and Limitations 74

4.6 Conclusion 75

Chapter 5: Sampling Textures with Missing Derivatives 77

5.1 Introduction 77

5.2 Texture Coordinate Derivatives at Visible Points 78

5.3 Further Applications 84

5.4 Comparison 85

5.5 Conclusion 85

Chapter 6: Differential Barycentric Coordinates 89

6.1 Background 89

6.2 Method 91

6.3 Code 92

Chapter 7: Texture Coordinate Gradients Estimation for Ray Cones 95

7.1 Background 95

7.2 Ray Cone Gradients 97

7.3 Comparison and Results 99

7.4 Sample Code 102

7.5 Conclusion 102

Chapter 8: Reflection and Refraction Formulas 105

8.1 Reflection 105

8.2 Refraction 105

Chapter 9: The Schlick Fresnel Approximation 109

9.1 Introduction 109

9.2 The Fresnel Equations 109

9.3 The Schlick Approximation 110

9.4 Dielectrics vs. Conductors 112

9.5 Approximations for Modeling the Reflectance of Metals 112

Chapter 10: Refraction Ray Cones for Texture Level of Detail 115

10.1 Introduction 115

10.2 Our Method 117

10.3 Results 121

10.4 Conclusion 124

Chapter 11: Handling Translucency with Real-Time Ray Tracing 127

11.1 Categories of Translucent Material 127

11.2 Overview 128

11.3 Single Translucent Pass 129

11.4 Pipeline Setup 131

11.5 Visibility for Semitransparent Materials 132

11.6 Conclusion 137

Chapter 12: Motion Blur Corner Cases 139

12.1 Introduction 139

12.2 Dealing with Varying Motion Sample Counts 141

12.3 Combining Transformation and Deformation Motion 144

12.4 Incoherent Motion 146

12.5 Conclusion 150

Chapter 13: Fast Spectral Upsampling of Volume Attenuation Coefficients 153

13.1 Introduction 153

13.2 Proposed Solution 155

13.3 Results 157

13.4 Conclusion 157

Chapter 14: The Reference Path Tracer 161

14.1 Introduction 161

14.2 Algorithm 163

14.3 Implementation 164

14.4 Conclusion 185

PART II: APIs and Tools 191

Chapter 15: The Shader Binding Table Demystified 193

15.1 The Shader Binding Table 193
15.2 Shader Record Index Calculation 196
15.3 API-Specific Details 198
15.4 Common Shader Binding Table Configurations 203
15.5 Summary 210

Chapter 16: Introduction to Vulkan Ray Tracing 213

16.1 Introduction 213
16.2 Overview 214
16.3 Getting Started 215
16.4 The Vulkan Ray Tracing Pipeline 215
16.5 HLSL/GLSL Support 218
16.6 Ray Tracing Shader Example 220
16.7 Overview of Host Initialization 223
16.8 Vulkan Ray Tracing Setup 224
16.9 Creating Vulkan Ray Tracing Pipelines 245
16.10 Shader Binding Tables 248
16.11 Ray Dispatch 252
16.12 Additional Resources 253
16.13 Conclusion 254

Chapter 17: Using Bindless Resources with DirectX Raytracing 257

17.1 Introduction 257
17.2 Traditional Binding with DXR 259
17.3 Bindless Resources in D3D12 261
17.4 Bindless Resources with DXR 267

17.5 Practical Implications of Using Bindless Techniques ... 273
17.6 Upcoming D3D12 Features ... 277
17.7 Conclusion ... 278

Chapter 18: WebRays: Ray Tracing on the Web ... 281

18.1 Introduction ... 281
18.2 Framework Architecture ... 283
18.3 Programming with WebRays ... 286
18.4 Use Cases ... 292
18.5 Conclusions and Future Work ... 298

Chapter 19: Visualizing and Communicating Errors in Rendered Images ... 301

19.1 Introduction ... 301
19.2 ꟻLIP ... 303
19.3 The Tool ... 307
19.4 Example Usage and Output ... 308
19.5 Rendering Algorithm Development and Evaluation ... 314
19.6 Appendix: Mean versus Weighted Median ... 318

PART III: Sampling ... 325

Chapter 20: Multiple Importance Sampling 101 ... 327

20.1 Direct Light Estimation ... 327
20.2 A Path Tracer with MIS ... 335
20.3 Closing Words and Further Reading ... 335

Chapter 21: The Alias Method for Sampling Discrete Distributions ... 339

21.1 Introduction ... 339
21.2 Basic Intuition ... 339

21.3 The Alias Method 341
21.4 Alias Table Construction 341
21.5 Additional Reading and Resources 342

Chapter 22: Weighted Reservoir Sampling: Randomly Sampling Streams 345

22.1 Introduction 345
22.2 Usage in Computer Graphics 346
22.3 Problem Description 346
22.4 Reservoir Sampling with or without Replacement 346
22.5 Simple Algorithm for Sampling with Replacement 347
22.6 Weighted Reservoir Sampling for $K > 1$ 348
22.7 An Interesting Property 348
22.8 Additional Reading 348

Chapter 23: Rendering Many Lights with Grid-Based Reservoirs 351

23.1 Introduction 351
23.2 Problem Statement 352
23.3 Grid-Based Reservoirs 356
23.4 Implementation 357
23.5 Results 363
23.6 Conclusions 364

Chapter 24: Using Blue Noise for Ray Traced Soft Shadows 367

24.1 Introduction 367
24.2 Overview 369
24.3 Blue Noise Samples 369
24.4 Blue Noise Masks 372
24.5 Void and Cluster Algorithm 374

24.6 Blue Noise Filtering 376
24.7 Blue Noise for Soft Shadows 378
24.8 Comparison with Interleaved Gradient Noise 389
24.9 Perceptual Error Evaluation 391
24.10 Conclusion 392

PART IV: Shading and Effects 399

Chapter 25: Temporally Reliable Motion Vectors for Better Use of Temporal Information 401

25.1 Introduction 401
25.2 Background 402
25.3 Temporally Reliable Motion Vectors 403
25.4 Performance 414
25.5 Conclusion 415

Chapter 26: Ray Traced Level of Detail Cross-Fades Made Easy 417

26.1 Introduction 417
26.2 Problem Statement 420
26.3 Solution 421
26.4 Future Work 423
26.5 Conclusion 424

Chapter 27: Ray Tracing Decals 427

27.1 Introduction 427
27.2 Decal Formulation 428
27.3 Ray Tracing Decals 429
27.4 Decal Sampling 435
27.5 Optimizations 435
27.6 Advanced Features 437

27.7 Additional Notes 438
27.8 Performance 438
27.9 Conclusion 440

Chapter 28: Billboard Ray Tracing for Impostors and Volumetric Effects 441

28.1 Introduction 441
28.2 Impostors 441
28.3 Volumetric Effects 446
28.4 Evaluation 452
28.5 Conclusion 453

Chapter 29: Hybrid Ray Traced and Image-Space Refractions 457

29.1 Introduction 457
29.2 Image-Space Refractions 458
29.3 Hybrid Refractions 459
29.4 Implementation 461
29.5 Results 464
29.6 Conclusion 466

Chapter 30: Real-Time Ray Traced Caustics 469

30.1 Introduction 469
30.2 Adaptive Anisotropic Photon Scattering 471
30.3 Ray-Guided Water Caustics 486
30.4 Conclusion 496

Chapter 31: Tilt-Shift Rendering Using a Thin Lens Model 499

31.1 Introduction 499
31.2 Thin Lens Model 500
31.3 Lens Shift 504

31.4 Lens Tilt 506
31.5 Directing the Tilt 507
31.6 Results 511

PART V: Intersection 517

Chapter 32: Fast and Robust Ray/OBB Intersection Using the Lorentz Transformation 519

32.1 Introduction 519
32.2 Definitions 520
32.3 Ray/AABB Intersection 521
32.4 Ray/OBB Intersection 523
32.5 Computing Additional Intersection Data 525
32.6 Conclusion 526

Chapter 33: Real-Time Rendering of Complex Fractals 529

33.1 Overview 529
33.2 Distance Functions 532
33.3 Implementation 537
33.4 Conclusion 542

Chapter 34: Improving Numerical Precision in Intersection Programs 545

34.1 The Problem 545
34.2 The Method 546

Chapter 35: Ray Tracing of Blobbies 551

35.1 Motivation 551
35.2 Anisotropic Blobbies 553
35.3 BVH and Higher-Order Motion Blur 554
35.4 Intersection Methods 556
35.5 Results 565

Chapter 36: Curved Ray Traversal 569

36.1 Introduction 569

36.2 Background 571

36.3 Implementation 578

36.4 Conclusions 594

Chapter 37: Ray-Tracing Small Voxel Scenes 599

37.1 Introduction 599

37.2 Assets 600

37.3 Geometry and Acceleration Structures 600

37.4 Shading 603

37.5 Performance Tests 606

37.6 Discussion 608

PART VI: Performance 613

Chapter 38: CPU Performance in DXR 615

38.1 Introduction 615

38.2 The Ray Tracing Pipeline State Object 615

38.3 The Shader Table 617

38.4 The Acceleration Structure 619

38.5 Conclusion 623

Chapter 39: Inverse Transform Sampling Using Ray Tracing Hardware 625

39.1 Introduction 625

39.2 Traditional 2D Texture Importance Sampling 627

39.3 Related Works 630

39.4 Ray Traced Inverse Transform Sampling 630

39.5 Implementation Details 633

39.6 Evaluation 635
39.7 Conclusion and Future Work 640

Chapter 40: Accelerating Boolean Visibility Operations Using RTX Visibility Masks 643

40.1 Background 643
40.2 Overview 644
40.3 Partial Visibility 645
40.4 Traversal 645
40.5 Visibility Masks as Boolean Visibility Functions 647
40.6 Accelerated Expressions 649
40.7 Solid Caps 651
40.8 Camera Initialization 656

Chapter 41: Practical Spatial Hash Map Updates 659

41.1 Introduction 660
41.2 Spatial Hashing 660
41.3 Complex Data Storage and Update 664
41.4 Implementation 665
41.5 Applications 668
41.6 Conclusion 670

Chapter 42: Efficient Spectral Rendering on the GPU for Predictive Rendering 673

42.1 Motivation 673
42.2 Introduction to Spectral Rendering 675
42.3 Spectral Rendering on the GPU 679
42.4 Multiplexing with Semitransparent Materials 685
42.5 A Step Toward Real-Time Performance 689

42.6 Discussion 693
42.7 Conclusion and Outlook 696

Chapter 43: Efficient Unbiased Volume Path Tracing on the GPU 699

43.1 Background 700
43.2 Compressed Data Structure 702
43.3 Filtering and Range Dilation 703
43.4 DDA Traversal 704
43.5 Results 707
43.6 Conclusion 710

Chapter 44: Path Tracing RBF Particle Volumes 713

44.1 Introduction 714
44.2 Overview 715
44.3 Implementation 717
44.4 Results and Conclusion 719

Chapter 45: Fast Volumetric Gradient Shading Approximations for Scientific Ray Tracing 725

45.1 Introduction 725
45.2 Approach 727
45.3 Results 728
45.4 Conclusion 732

PART VII: Ray Tracing in the Wild 737

Chapter 46: Ray Tracing in Control 739

46.1 Introduction 739
46.2 Reflections 742
46.3 Transparent Reflections 745

46.4 Near Field Indirect Diffuse Illumination 749
46.5 Contact Shadows 750
46.6 Denoising 753
46.7 Performance 761
46.8 Conclusions 763

Chapter 47: Light Sampling in Quake 2 Using Subset Importance Sampling 765

47.1 Introduction 765
47.2 Overview 767
47.3 Background 768
47.4 Stochastic Light Subset Sampling 771
47.5 Reducing Variance with Pseudo-marginal MIS 777
47.6 Stochastic Light Subset MIS 781
47.7 Results and Discussion 783
47.8 Conclusions 786

Chapter 48: Ray Tracing in Fortnite 791

48.1 Introduction 791
48.2 Goals 792
48.3 Challenges 793
48.4 Technologies 794
48.5 Fortnite Cinematics 819
48.6 Conclusion 819

Chapter 49: ReBLUR: A Hierarchical Recurrent Denoiser 823

49.1 Introduction 823
49.2 Definitions and Acronyms 825
49.3 The Principle 825

49.4 Inputs 826
49.5 Pipeline Overview 827
49.6 Disocclusion Handling 831
49.7 Diffuse Accumulation 831
49.8 Specular Accumulation 832
49.9 Sampling Space 839
49.10 Spatial Filtering 840
49.11 Anti-lag 841
49.12 Limitations 842
49.13 Performance 842
49.14 Future Work 843

Chapter 50: Practical Solutions for Ray Tracing Content Compatibility in Unreal Engine 4 845

50.1 Introduction 846
50.2 Hybrid Translucency 846
50.3 Foliage 854
50.4 Summary 858

Preface

In 2018, real-time ray tracing arrived in consumer GPU hardware and swiftly established itself as a key component of how images would be generated moving forward. Now, three years later, the second iteration of this hardware is available, mainstream game consoles support ray tracing, and cross-platform API standards have been established to drive even wider adoption. Developers and researchers have been busy inventing (and reinventing) algorithms to take advantage of the new possibilities created by these advancements. The "gems" of knowledge discovered during this process are what you will find in the pages that follow.

Before diving into the treasure trove, we want to make sure you know the following:

- Supplementary code and other materials related to this book can be found linked at http://raytracinggems.com.
- All content in this book is open access.

Open access content allows you to freely copy and redistribute any chapter, or the whole book, as long as you give appropriate credit and you are not using it for commercial purposes. The specific license is the Creative Commons Attribution 4.0 International License (CC-BY-NC-ND).[1] We put this in place so that authors, and everyone else, can disseminate the information in this volume as quickly, widely, and equitably as possible.

Writing an anthology style book, such as this, is an act of collective faith. Faith that the authors, editors, publisher, and schedules all converge to produce a worthwhile result for readers. Under normal circumstances this is a difficult task, but the past year and a half has been anything but "normal." This book's call for participation was posted in late November 2019, just as the COVID-19 virus began to emerge in Asia. In the 18 months since, a once-in-a-century pandemic has surged. Its impact has touched every person on Earth—and it is not yet over. The virus has taken lives, battered livelihoods, and broken the way of life as we knew it. We postponed this book with the quiet conviction

[1] https://creativecommons.org/licenses/by-nc-nd/4.0/.

that the coronavirus would be overcome. As time passed, the development of vaccines made breakneck progress, remote work in quarantine became a new staple of life, and authors began to write about computer graphics again. As a result, this book was written entirely during quarantine. We sincerely thank the authors for their passion and dedication—and faith—all of which made this book possible under extraordinary circumstances.

Further thank-yous are in order for the individuals whose support made this book possible. Without a doubt, this book would not have happened without Jaakko Haapasalo, Charlotte Byrnes, Eric Haines, and Tomas Akenine-Möller. Thank you to Ankit Patel, Ethan Einhorn, Peter Harrison, Fredrik Liljegren, Eric Reichley, and Ilkka Koho for their efforts on financing, marketing, and coordinating with partners.

We are especially grateful to Natalie Pao, Susan McDermott, and the entire team at Apress for their resilience as the ground shifted beneath us. We thank Steven Hendricks and the team at Fineline Graphics & Design for their swift and high-quality work on figures throughout the book.

We thank Craig Hynes, Alexey Panteleev, Amanda Lam, and Jennifer Reyburn for their work on the cover designs, interior spreads, and visual design of the entire book. The NVIDIA Creative team that created *Marbles at Night* (featured on the cover) includes Gavriil Klimov (Creative Director), Jacob Norris (Lead Environment Artist), Andrej Stefancik (Senior 3D Artist), Gregor Kopka (Lead 3D Artist), Artur Szymczak (Senior Lighting Artist), Chase Telegin (Technical Artist), Alessandro Baldasseroni (Lead 3D Artist), Fred Hooper (Lead VFX Artist), and Ilya Shelementsev (Senior 3D Artist).

We extend a special thank-you to our partners who generously created content for the special edition covers. These include the following:

1. Ubisoft (*Watch Dogs: Legion*): Patrick Ingoldsby, Fotis Prasinis, Matthew Mackillop, Kenny Lam, and team.
2. Epic Games (*Fortnite*): Paul Oakley, Scott James, Jon Kiker, Laura Schreiber, and team.
3. Remedy Games (*Control*): Tatu Aalto, Mikko Orrenmaa, and team.
4. id Software (*Quake II RTX*).

Last, but not least, major credit and thanks are due to our excellent team of section editors—Per Christensen, David Hart, Thomas Müller, Jacob Munkberg, Angelo Pesce, Josef Spjut, Michael Vance, and Cem Yuksel—for their careful reviewing, editing, and guidance.

—Adam Marrs, Peter Shirley, and Ingo Wald
June 2021

Foreword

by Andrew Glassner and Ignacio Llamas

Think globally, act locally. This idea was introduced by town planner Patrick Geddes in 1915 [4]. This principle leads me to buy the things I need from local, independent stores, and to prefer food that has been grown nearby. Over a hundred years since this idea emerged, it's still guiding not just me, but also social activists, environmentalists, and people making images with computers.

Prior to Jim Kajiya's seminal rendering equation paper in 1986 [3], image rendering was a growing collection of algorithms that generally each bit the apple in a different way. The rendering equation neatly organized and described the two essential operations at the heart of image rendering: finding the light arriving at a point, and computing the light leaving that point in a given direction. The information needed to solve the equation was gathered globally, from the entire environment, and the final solution was computed locally, at the point being shaded. That is, gather globally, shade locally.

An important feature of the rendering equation for us, the wise readers of *Ray Tracing Gems II*, is that it is an integral equation that can be solved numerically using a collection of point samples. When we think about the problem of finding the light incident at a point, ray tracing is the obvious and elegant way to find those samples.

Wait, that's not fair. *Obvious* is one of those words that implies that all the smart people would think of any given thing immediately and they'd all agree. Calling something "obvious" implies that if you didn't immediately think of exactly that and agree with everyone else, then you're not smart. *Obvious* is a punishment word, like *intuitive*.

Ray tracing as we know it in this book would not be obvious to the ancient Greeks, and they were as smart as anyone around today. Consider the Greek philosopher Empedocles, who was born around 440 BCE. He's the fellow who came up with the theory that all objects are made up of earth, air, fire, and

water. Because everything is formed by some mixture (or change in the mixture) of these elements, something had to do that mixing and changing. That work was taken on by two additional forces: love, which brought the elements together, and strife, which pulled them apart.

Less well known today is Empedocles' theory of light, now called *emission theory* (or extramission theory). The key idea of emission theory asserts that our ability to see the world around us begins with a burning fire that we each carry inside our head. When our eyelids are open, this firelight escapes in the form of rays shooting out of our eyes, illuminating the world before us [2]. Essentially, emission theory holds that we are human flashlights. The light we beam out into the environment bounces off of the objects around us and back into our eyes, and thus we behold the world. Plato believed in emission theory, and even Ptolemy's famous book *Optics* described phenomena as complex as refraction using rays of light coming from the eyes.

It's easy for us to refute emission theory in any number of ways today. (Challenge: name three ways to disprove emission theory in 30 seconds. Ready? Go!) But that's in retrospect. At the time, the theory made sense to people, and despite Euclid expressing some doubts, emission theory was the accepted explanation of vision for at least a thousand years. I find it startling that roughly half of the people alive today believe in emission theory [6].

The impact of emission theory on western culture can still be seen in today's language and entertainment. A common idiom in American English describes someone feeling a strong, joyful emotion by saying that "their eyes lit up," as though their passion had stoked their internal fire, causing their eyes to glow. The title character in the *Superman* comics has "heat vision," a superpower that lets him make anything extremely hot just by staring at it. The comics show this power with red beams shooting outward from his eyes, making his target hot enough to smoke, or even erupt in flames. All thanks to the rays coming out of his eyes, carrying the heat of his inner super-fire.

Given all of the evidence for the Sun as our primary source of light, and the flaws in emission theory, why has that theory dominated Western thinking for so long? One reason is that before the Enlightenment and the development of the scientific method, appeal to authority was an acceptable means for settling almost any kind of argument. For example, in about 350 BCE Aristotle stated that, "Males have more teeth than females in the case of men, sheep, goats, and swine …" [1]. For centuries thereafter, Aristotle's opinions were

still so influential that people believed that men had more teeth than women. Despite farmers and doctors constantly looking into the mouths of people and animals, nobody felt the need to bother counting the teeth they found there, or announcing what they found if they did. If Aristotle said that men have more teeth than women, then they do. In the same way, Empedocles told everyone that we see things by beaming light outward from our eyes, so that settled that.

An appealing feature of emission theory is its efficiency. The fire in our heads lights up the things we need to see, and nothing else. This seems like a sensible use of an important resource. After all, why would you waste valuable head firelight on things you're not looking at? All other matters aside, the efficiency argument is a good one. Firelight is precious, and only so much of it can fit inside one's head, so it makes sense to be as frugal with it as possible.

This efficiency is at the heart of Turner Whitted's classic paper that introduced modern ray tracing [5], which brilliantly combined the minimal number of rays required by emission theory with the transport of light described by modern optics. When we generate eye rays (and rays from a point to be shaded), we're pretending to practice emission theory, but then we use modern optics to bring the light back to carry out the shading step.

This marriage of the ancient and the modern approaches to light brings us to the newest of the new: the cutting-edge ray tracing algorithms in this book. From its humble beginnings as a thought experiment almost 2500 years ago, we're now using ray tracing to simulate rays of light using GPUs with billions of transistors, each only a few nanometers across. Elegant, powerful, and capable of generating images both beautiful and accurate, ray tracing is once again ascendant. What was old is new again, and it works better than ever.

With the tools in this book, I hope you'll be able to keep the fire in your head well stoked with great ideas and positive emotions, and to send rays into the environment that let you create beautiful and meaningful work as you gather globally, and shade locally.

—Andrew Glassner
Senior Research Scientist, Weta Digital
January 2021

* * *

By the time this book releases, it will have been over three years since we (NVIDIA) launched RTX at GDC 2018. The first RTX GPUs, based on the Turing architecture, arrived soon thereafter, making hardware-accelerated real-time ray tracing available for the first time ever on widely available consumer GPUs.[2] At the time we announced RTX at GDC 2018, we were running the *Star Wars* "Reflections" real-time ray tracing demo on a $68,000 DGX Station with four NVIDIA Volta GPUs (our most powerful GPUs at the time). As a result, it was not surprising that the technology was met with reasonably high levels of skepticism. What was perhaps more surprising to us was the relatively nonexistent speculation about what would come a few months later in our next GPU architecture launch. Clearly, the world was not quite ready for real-time ray tracing, and few gave much thought to the possibility of real-time ray tracing making it into games so quickly after.

As with any new technology, there is an adoption phase before widespread use takes hold. Our past experiences introducing new GPU hardware features told us that key new features can take about five years to gain widespread adoption (and it usually takes a few more years for these to trickle down to GPUs in mobile devices). Ray tracing, however, was quite different from the many previous features in the 25-year history of consumer GPUs. This was perhaps the most significant technological advance in GPUs since programmable shading was introduced 15 years earlier. The ray tracing capability we introduced enabled real-time rendering developers, for the first time, to think about rendering algorithms in radically new, yet more natural, ways. Real-time ray tracing opened up a world of algorithmic choices that were previously confined to offline rendering, which had started the transformation from rasterization techniques and physically plausible rendering hacks to physically based rendering with ray tracing about a decade earlier. We believed this presaged what we were about to see in the real-time rendering world—and we were not wrong.

Because ray tracing provides such a radical new tool, it gathered a level of interest we rarely see among the top rendering developers. This allowed us to have a significant showing at launch with several engines and over ten games showing ray traced visuals. Nevertheless, this was still early days for all of us

[2] Imagination Technologies had ray tracing hardware designs, but these did not become available in widely used consumer products.

working on real-time ray tracing. Despite the relative success of this launch event, it was difficult to convey to the average person the significance of the technological advance we were in the midst of.

In the three years since Turing's reveal, there has been a growing understanding of just how transformative real-time ray tracing is. It not only enables higher-quality graphics in games, but also improves workflows across many industries that are increasingly relying on digital models and virtual simulations. By rendering physically based photorealistic imagery in real time, without long precomputation times, we unlock faster iterative workflows in architecture, product design, animation, and visual effects. Increasingly, we are seeing real-time ray traced rendering become central to robotics and autonomous driving, where accurate simulations of the physical world are used to train AI-based algorithms that are powering the next technological revolution. All of these use cases rely on ray tracing for the simulation of light transport, but ray tracing is also involved in lesser-known applications including the simulation of other physical phenomena, such as sound wave and electromagnetic wave propagation. These are also going to become increasingly important across various industries.

By now you may be wondering why I am providing this brief personal perspective of how real-time ray tracing was introduced to the world. If you are reading this book, you probably know something about ray tracing and rendering already. So, as you are spending time reading this foreword, there is one message I want you to take away from it. I want to reaffirm your belief in ray tracing. I want you to see how impactful real-time ray tracing can be, and with that hopefully motivate you to learn as much from this book as you can. I truly believe real-time ray tracing is one of those enabling technologies that has the power to create ripples across many fields. Ray tracing is here to stay, and it will become as widely used across the world in the next few years as mobile phones are. It is for this reason that this series of books, *Ray Tracing Gems*, is now in its second iteration. *Ray Tracing Gems* exists to gather and distribute the ever-increasing knowledge earned from our collective experiences—and ultimately help make the impact of our work more significant to the world.

Our journey is still just beginning. There are many things yet to be done to improve what we can do with real-time ray tracing. As with any tool, ray tracing must be used wisely. We must trace those rays that are most useful. Sampling algorithms help us do this. In addition, ray tracing alone is not

sufficient to generate beautiful images. It must be combined with complex physically based material models, backed by quality content, and executed efficiently in the presence of the wide variety inherent to the real world. Furthermore, because rendering algorithms using ray tracing are usually based on stochastic sampling with low sample budgets, some noise often remains even with the best sampling algorithms. Smart denoising algorithms are needed to make real-time ray tracing feasible. Finally, ray tracing needs something to trace rays against. The geometric and volumetric models of the world must be stored in efficient acceleration structures that are fast to update.

These are just a few of the key high-level challenges in front of us. There are many specific problem areas within these challenges that deserve full chapters or even PhD dissertations. You will find a curated selection of solutions to such problems in this book. I hope you find the chapters within useful and inspiring, and that they make your ray tracing experience that much more satisfying.

—Ignacio Llamas
Omniverse RTX Lead, NVIDIA
January 2021

REFERENCES

[1] Aristotle. *The History of Animals, Book II, Part 3*. Translated by D. W. Thompson. 350 BCE. http://classics.mit.edu/Aristotle/history_anim.2.ii.html.

[2] Emission theory. *Wikipedia*, https://en.wikipedia.org/wiki/Emission_theory_(vision). Accessed January 2021.

[3] Kajiya, J. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, pages 143–150, 1986. DOI: 10.1145/15922.15902.

[4] Think globally, act locally. *Wikipedia*, https://en.wikipedia.org/wiki/Think_globally,_act_locally. Accessed January 2021.

[5] Whitted, T. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[6] Winer, G. A., Cottrell, J. E., Gregg, V., Fournier, J. S., and Bica, L. A. Fundamentally misunderstanding visual perception: Adults' beliefs in visual emissions. *American Psychologist*, 57(6–7):417–424, 2002.

CONTRIBUTORS



Tatu Aalto is lead graphics programmer at Remedy Entertainment. He has been working in the games industry since 2003 and has a background in real-time graphics and software rasterizing before that. Tatu has spent most of his industry career on building rendering technology and has given lectures on computer graphics at various conferences. His current interests include ray tracing, modern rendering techniques, and advances in photorealistic rendering and image quality.



Maksim Aizenshtein is a Senior Manager at NVIDIA, USA. He leads a team of developers who are developing the RTX Renderer, which is a part of the Omniverse ecosystem. His team mainly focuses on researching, developing, and implementing new algorithms in 3D graphics, in particular in the field of ray tracing. His previous positions include leading the 3DMark team in UL Benchmarks, overseeing and developing graphics benchmarks as well as one of the first applications to utilize the DirectX Raytracing API. Before UL Benchmarks, at Biosense-Webster he was responsible for developing GPU-based rendering in modern medical imaging systems. Maksim received his BSc in computer science from Israel's Institute of Technology in 2011.



Tomas Akenine-Möller is a distinguished research scientist at NVIDIA, currently on leave from his position as professor in computer graphics at Lund University. Tomas coauthored *Real-Time Rendering* and *Immersive Linear Algebra* and coedited *Ray Tracing Gems*.



Pontus Andersson is an industrial PhD student at NVIDIA, affiliated with the Centre for Mathematical Sciences at Lund University, Sweden, and part of the Wallenberg AI, Autonomous Systems and Software Program. He holds an MSc in engineering mathematics from Lund University. His research interests span computer graphics, image analysis, and human perception.



Dirk Gerrit van Antwerpen is a senior software engineer at NVIDIA in Berlin. He received his MS degree in computer science from Delft University of Technology, writing his graduate thesis on the topic of physically based rendering on the GPU. During his graduate work, Dirk co-developed the Brigade real-time rendering engine, subsequently acquired by OTOY. He continued working as a postgraduate researcher at the agricultural university of Wageningen on light transport for plant growth simulations. He joined NVIDIA in 2012 and has since been heavily involved in the development of the NVIDIA Iray renderer and the NVIDIA OptiX ray tracing API. Dirk is an expert in physically based light transport simulation and parallel computing.



Wessam Bahnassi is a 20-year veteran in 3D engine design and optimization. His latest published games include *Batman: Arkham Knight*, and his own native 120-FPS PlayStation VR space shooter game *Hyper Void*. He is a contributor and section editor in the *ShaderX*, *GPU Pro*, and *GPU Zen* series of books. His current work at NVIDIA includes rendering optimizations and contributing to a couple of the numerous cool research projects there.



Neil Bickford is a developer technology engineer at NVIDIA, where he wrote the vk_mini_path_tracer Vulkan ray tracing tutorial and built the tone mapping algorithms for *Quake II* RTX and *Minecraft* RTX. He is also the main developer for the NVIDIA Texture Tools Exporter, a widely used frontend for NVTT GPU texture compression.



Trevor David Black is a software engineer in the Android Games and Graphics group at Google in California. Prior to joining Google, Trevor worked as a graphics engineer at Intel. He is currently best known for his work as one of two primary editors of the *Ray Tracing in One Weekend* series. His research interests include color science, physically based rendering, and computational geometry. He received his MS degree in electrical engineering from the University of California, Los Angeles, in 2018.



Jakub Boksansky is a DevTech engineer at NVIDIA working on delivering ray tracing technology to game titles. His professional interests include ray tracing–based algorithms and advanced real-time rendering techniques. Jakub found his interest in computer graphics while developing web-based computer games using Flash, and he later developed and published several image effect packages for the Unity game engine. Prior to joining NVIDIA, he was a research scientist at Czech Technical University Prague, where he worked on improving the performance of CUDA-based path tracers.



Carson Brownlee is a graphics engineer in Intel's oneAPI Rendering Toolkit group working on large-scale, parallel, and distributed ray tracing problems in the fields of scientific visualization and studio rendering. Prior to joining Intel, he researched visualization and ray tracing at the Texas Advanced Computing Center at the University of Texas at Austin. He received a PhD in computer science from the University of Utah and a BS in computer science from Gonzaga University.



Felix Brüll is a PhD student at the Technical University of Clausthal, Germany. His current research topic is the acceleration and improvement of global illumination with current ray tracing hardware. Felix received a BS in computer science and an MS in computer science from the University of Clausthal in 2018 and 2020, respectively.



Juan Cañada is a lead engineer at Epic Games, where he leads ray tracing development in the Unreal Engine engineering team. Before, Juan was head of the Visualization Division at Next Limit Technologies, where he led the Maxwell Render team for more than 10 years. He also was a teacher of data visualization and big data at the IE Business School.



Joaquim Bento Cavalcante-Neto is a full professor in the Computing Department at the Federal University of Ceará in Brazil. He received his PhD from Pontifical Catholic University of Rio de Janeiro (PUC-Rio) in 1998. His PhD was in civil engineering, and he worked mainly with applied computer graphics. During his PhD work, he spent one year as a visiting scholar (PhD student) at Cornell University (1996). He was also a postdoctoral associate (2003) and a visiting associate professor (2013) at the same university. His research interests include computer graphics, virtual reality, and animation, but also computer simulations, numerical methods, and computational systems.



Per Christensen is a principal scientist in Pixar's RenderMan group in Seattle. His main research interests are efficient ray tracing and global illumination in very complex scenes. He received an MSc degree in electrical engineering from the Technical University of Denmark and a PhD in computer science from the University of Washington. Before joining Pixar, he worked at Mental Images (now part of NVIDIA) in Berlin and at Square USA in Honolulu. His movie credits include every Pixar movie since *Finding Nemo*, and he has received an Academy Award for his contributions to efficient point-based global illumination and ambient occlusion.



Cyril Crassin is a senior research scientist at NVIDIA. His research interests and expertise include GPU-oriented methods for real-time realistic rendering, light transport, and global illumination, as well as efficient antialiasing and filtering techniques using alternative geometric and material representations—especially voxel-based representations. Prior to joining NVIDIA, Cyril obtained his PhD from Grenoble University at INRIA in 2011.



David DeMarle works at Intel on advancing ray tracing within the applied field of scientific visualization. He contributes primarily to the open source VTK, ParaView, and OSPRay projects. David enjoys the challenge of applying new techniques and new hardware to bring about new ways to understand data.



Marc Droske is Head of Rendering Research at Weta Digital. He received his PhD in applied mathematics at the Institute for Numerical Simulation, Duisburg, where he focused on differential geometry, inverse problems, variational methods, and partial differential equations. After a postdoctoral year at the University of California, Los Angeles, he joined Mental Images (later NVIDIA ARC), where he worked on geometry processing algorithms and later on GPU-accelerated rendering algorithms using CUDA and the earliest versions of OptiX. He joined Weta in 2014 to work on their in-house production renderer Manuka, where he developed rendering algorithms and designed and wrote large parts of the path sampling framework used in final-frame rendering.



Alex Evans was a cofounder and technical director of MediaMolecule, a game studio best known for its work in "creative gaming," with titles including *LittleBigPlanet* and *Dreams*. He joined NVIDIA Research at the end of 2020.



Alban Fichet is a postdoctoral researcher at Charles University in Prague. Prior, he worked at Institut d'Optique Graduate School in Bordeaux, where he participated in the development of the Malia Rendering Framework, a GPU-accelerated spectral rendering toolkit. His current research focuses on spectral and bispectral modeling and rendering. He received his PhD in 2019 from Inria Grenoble on efficient representations for measured reflectance.



Pascal Gautron's work at NVIDIA is focused on designing and optimizing fast, high-quality rendering solutions. Over the last 18 years, he has gathered an academic and industrial background in computer graphics research, photorealistic image synthesis, real-time rendering, and movie post-production.



Anastasios Gkaravelis is a postdoctoral fellow in the Department of Informatics of the Athens University of Economics and Business. He received his PhD in computer science from the same institution in 2020. His work has been published in high-ranking journals and international conferences, and his main research interests focus on lighting design optimization methods, photorealistic rendering, real-time path tracing, and global illumination. Anastasios is a cofounder of Phasmatic, a company that aims to offer photorealistic rendering solutions on the Web.



Christiaan Gribble is the Director of High Performance Computing and a Principal Research Scientist within the Applied Technology Operation at SURVICE Engineering Company. His research explores the synthesis of interactive visualization and high-performance computing. Gribble received a PhD in computer science from the University of Utah in 2006.



Holger Gruen started his career in three-dimensional real-time graphics over 25 years ago writing software rasterizers. In the past he has worked for game middleware, game, military simulation, and GPU hardware companies. He currently works within Intel's XPU Technology and Research Group.



Eric Haines currently works at NVIDIA on interactive ray tracing. He coauthored the books *Real-Time Rendering* and *An Introduction to Ray Tracing*, coedited *Ray Tracing Gems* and *The Ray Tracing News*, and cofounded the *Journal of Graphics Tools* and the *Journal of Computer Graphics Techniques*.



Johannes Hanika was introduced to the secrets of computer graphics in Ulm University, working on low-level ray tracing optimisation, quasi-Monte Carlo point sets, light transport simulation, and image denoising. For the following 10 years, he worked as a researcher for Weta Digital, first in Wellington, New Zealand, and later remotely from Germany. He is a co-architect of Manuka, Weta Digital's physically based spectral renderer. Since 2013, Johannes has worked as a postdoctoral fellow at the Karlsruhe Institute of Technology. In 2009, he founded the darktable open source project, a workflow tool for RAW photography. Johannes has a movie credit for *Abraham Lincoln: Vampire Hunter*.



David Hart is a researcher and engineer on the NVIDIA OptiX team. Before that, he spent 15 years in production making CG films and games for DreamWorks and Disney. He founded a company making an online multi-user WebGL whiteboard, and sometimes he works on the side as an amateur digital artist using artificial evolution techniques. David is a co-inventor on patents for sampling and digital hair styling. His goal is to make beautiful pictures using computers and build great tools to make it faster and easier. David holds an MS from the Program of Computer Graphics at Cornell University.



Evan Hart is a Principal Engineer in NVIDIA's Developer Technology group. He has spent more than 20 years working with application developers to improve the quality and performance of real-time 3D graphics. Most recently, his focus has been on all things ray tracing in the Unreal Engine.



Nikolai Hofmann is a PhD student at Friedrich-Alexander University Erlangen-Nüremberg, Germany. His research interests mainly revolve around (volume) ray/path tracing, deep learning, their intersection, and the efficient implementation thereof. He received an MS in computer science from the Friedrich-Alexander University in 2020 and interned with the real-time rendering research group at NVIDIA.



Thiago Ize is a principal software engineer at Autodesk. He has worked on the Arnold renderer since 2011 when he joined Solid Angle, which was later acquired by Autodesk. His interests include improving performance, robustness, and memory usage. In 2017 he received a Scientific and Engineering Academy Award for his contributions to the Arnold renderer. He received his PhD in computer science from the University of Utah in 2009.



Johannes Jendersie has been a senior graphics software engineer at NVIDIA in Berlin since 2020. Before that, he received his PhD in computer graphics, with a focus on improving Monte Carlo light transport simulations with respect to robustness and parallelization, from the Technical University Clausthal, Germany. At NVIDIA he works on the Iray renderer, extending its simulation capabilities and working on noise reduction for difficult light scenarios.



Gregory P. Johnson is a senior graphics software engineer at Intel. He is working on high-performance, ray tracing–based visualization and rendering libraries. Before joining Intel, Greg was a research associate at the Texas Advanced Computing Center, focusing on large-scale scientific visualization and high-performance computing. He received a PhD in aerospace engineering from the University of Texas at Austin.



Paula Jukarainen is a developer technology engineer at NVIDIA. Recently, she has been focusing on implementing ray tracing effects in titles such as *Control* and *Minecraft* RTX. Prior to joining NVIDIA, she worked as a rendering engineer and team lead at UL (previously known as Futuremark), contributing to several 3DMark benchmarks. She holds an MSc Tech degree in computer science from Aalto University.



Patrick Kelly is a senior rendering programmer at Epic Games, working with real-time ray tracing on Unreal Engine. Before arriving at Epic Games, he spent nearly a decade in production rendering at studios such as DreamWorks Animation, Weta Digital, and Walt Disney Animation Studios. Patrick received a BS in computer science from the University of Texas at Arlington in 2004 and an MS in computing from the University of Utah in 2008.



Andrew Kensler is a senior graphics engineer on the Scout simulation team at Amazon. Prior to that, he was a senior software engineer on the core RenderMan team at Pixar Animation Studios, where he worked for more than 11 years. He received a PhD in computer science from the University of Utah, where his studies focused on ray tracing. His passion for graphics programming extends back to his childhood.



Oliver Klehm is a senior software engineer at NVIDIA. He works on core ray tracing functionality in OptiX to bring hardware-accelerated ray tracing to CUDA-based renderers. He is also part of the NVIDIA Iray team, focusing on efficient but highly realistic light transport algorithms. Previously, he graduated from the Max Planck Institute for Informatics in Saarbruecken, Germany, with a PhD in computer science and worked in the rendering team of DNEG, one of the largest European visual effects companies.



Aaron Knoll is a senior graphics software engineer at Intel, working on ray tracing applications in visualization and high-performance computing. He earned his PhD in computer science from the University of Utah and has worked at Argonne National Laboratory, the Texas Advanced Computing Center, and NVIDIA.



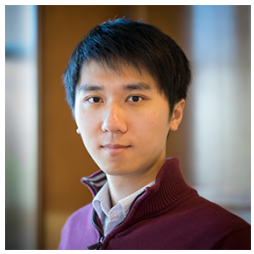
Christopher Kulla is a senior rendering engineer at Epic Games, where he focuses on ray tracing and light transport. He previously worked at Sony Pictures Imageworks for 13 years on a proprietary fork of the Arnold renderer, for which he was recognized with a Scientific and Engineering Award from the Academy of Motion Picture Arts and Sciences in 2017.



Dylan Lacewell is currently a Senior Development Engineer at NVIDIA and was also an OptiX developer technology engineer for several years. He has also worked for Apple, PDI/Dreamworks Animation, and Walt Disney Feature Animation. He first learned ray tracing as a graduate student at the University of Utah.



Anders Lindqvist is a senior engineer at NVIDIA in Lund, Sweden, working with ray tracing. He received an MS in engineering mathematics at Lund University in 2008. While previously a real-time person with roots in the demoscene, he discovered non-real-time rendering during a graphics course. After graduation he has worked with rendering in various forms at Illuminate Labs, Autodesk, and Rapid Images. A common theme has been light map baking for games, on the Beast games middleware at Illuminate Labs/Autodesk as well as in the Stingray games engine while at Autodesk.



Shiqiu Liu is a Senior Research Scientist at NVIDIA Applied Deep Learning Research, where he currently explores the application of deep learning in real-time graphics pipelines and other topics. In the last several years at NVIDIA, Shiqiu has worked in multiple roles and contributed to the early research and development of several next-geneneration rendering technologies, including deep learning–based reconstruction techniques like Deep Learning Super Sampling (DLSS) 2.0, real-time ray tracing and denoising, and virtual reality rendering. In his spare time, Shiqiu enjoys traveling and landscape and nightscape photography.



Hélio Lopes is an Associate Professor in the Department of Informatics at Pontifical Catholic University of Rio de Janeiro (PUC-Rio). He has a doctoral degree in mathematics (1996), a master's degree in informatics (1992), and a bachelor's degree in computer engineering (1990). All of these degrees were obtained at PUC-Rio. His research is mainly in the fields of data science and visualization. He likes to develop and apply advanced mathematical and computational techniques to solve real-world problems that have engineering, scientific, and industrial relevance. He is the principal investigator of the Data Science Laboratory of the Department of Informatics at PUC-Rio.



José Gilvan Rodrigues Maia has been both a PhD student and an associate professor at the Federal University of Ceará (UFC) since 2010. He has worked on a number of research projects ranging from computer graphics and virtual reality to computer vision and artificial intelligence. His current research interests include real-time rendering, collision detection, simulation, and computer vision. Gilvan spends some of his spare time having fun with coding, reading, and testing ideas from computer science in general.



Zander Majercik is a Research Scientist at NVIDIA. His research focuses on efficient approximations for real-time global illumination with a focus toward cloud graphics. His previous projects include novel near-eye display technologies, deep learning–based gaze tracking, and experiments in vision science and esports performance. He is a maintainer of the open source graphics engine G3D.



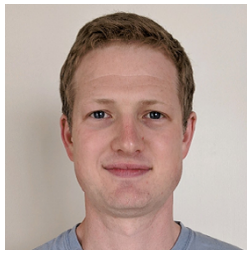
Adam Marrs is a Senior Graphics Engineer in the Game Engines and Core Technology group at NVIDIA, where he works on real-time rendering and ray tracing for games. His experience includes work on commercial game engines, shipped games, and published graphics research. Prior to NVIDIA, Adam discovered his passion for computer graphics while working in advertising on Flash-based websites and games. He received a BS in computer science from Virginia Tech and a PhD in computer science from North Carolina State University.



Morgan McGuire is the Chief Scientist at Roblox, channeling and accelerating innovation across the company to build a creative, safe, civil, and scalable Metaverse. Morgan is also known for previous work as research director at NVIDIA; professor at Williams College, University of Waterloo, and McGill University; coauthor of *The Graphics Codex*, *Computer Graphics: Principles and Practice*, and *Creating Games*; and chairperson of the EGSR, I3D, NPAR, and HPG conferences.



Johannes Meng is a graphics software engineer at Intel. His current work focuses on high-performance data structures for scalable volume rendering. Before Intel, he worked as a rendering researcher at Weta Digital. Johannes holds a PhD in computer science from Karlsruhe Institute of Technology.



Peter Morley works as a senior developer technology engineer at NVIDIA. Most of his work is focused on integrating DirectX Raytracing (DXR) into AAA game engines. His previous work includes implementing DXR 1.0 and 1.1 in AMD's driver stack. He completed his MSc at the University of Rhode Island in 2017 and helped research voxel space ray tracing using hidden Markov models for state estimation. When not flipping RTX On in games, he enjoys playing soccer and listening to metal.



Nate Morrical is a PhD student at the University of Utah, is an intern at NVIDIA, and is currently working under Valerio Pascucci as a member of the CEDMAV group at the Scientific Computing and Imaging Institute (SCI). His research interests include high-performance GPU computing, real-time ray tracing, and human-computer interaction. Prior to joining SCI, Nate received his BS in computer science from Idaho State University, where he researched interactive computer graphics and computational geometry under Dr. John Edwards.



Thomas Müller is a senior research scientist at NVIDIA, working on the intersection of light transport simulation and machine learning. Before joining NVIDIA, Thomas obtained his PhD from Eidgenössische Technische Hochschule (ETH) Zürich, where he collaborated with Disney, leading to several awards and patents in the offline rendering world. The resulting algorithms were implemented in multiple production renderers, including Pixar's RenderMan and Walt Disney Animation's Hyperion. Outside of research, Thomas created commercial anti-cheat software for the MMORPG *FlyFF*, developed several core components of the online rhythm game *osu!*, and maintains the image analysis tool "tev."



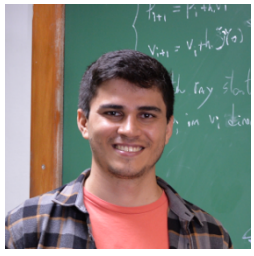
Jacob Munkberg is a senior research scientist in NVIDIA's real-time rendering research group. His current research focuses on machine learning for computer graphics. Prior to NVIDIA, he worked in Intel's Advanced Rendering Technology team and cofounded Swiftfoot Graphics, specializing in culling technology. Jacob received his PhD in computer science from Lund University and his MS in engineering physics from Chalmers University of Technology.



David Murray is currently working as a postdoctoral researcher at the Institut Optique Graduate School in Bordeaux. He received his PhD in 2018 from Bordeaux University, on real-time and legible visualization solutions for scientific applications. Until June 2020, he was a postdoctoral researcher at Inria Bordeaux-Sud-Ouest, working on path guiding for light transport. His current research topic focuses on spectral light transport and material appearance (acquisition and modeling) for cultural heritage. He is also an active developer of the Malia Rendering Framework, a GPU-accelerated spectral rendering toolkit.



Jim Nilsson is a graphics researcher with the Real-Time Rendering group at NVIDIA Research. His research interests include computer graphics, high-performance rendering, image quality assessment, image metrics, and visual computing. Jim holds a PhD in computer architecture from Chalmers University of Technology in Gothenburg, Sweden.



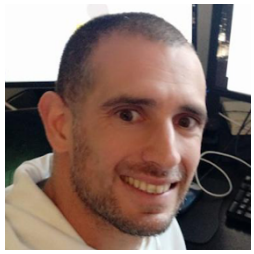
Tiago Novello is a postdoctoral researcher at National Institute for Pure and Applied Mathematics' Vision and Graphics (VISGRAF) Laboratory. He obtained a DSc in applied mathematics at Pontifical Catholic University of Rio de Janeiro, an MSc in mathematics from the Federal University of Alagoas, and a BSc in mathematics at State University of Paraná.



Yuriy O'Donnell is a graphics programmer at Epic Games, working on Unreal Engine, mostly focused on ray tracing. He previously worked on the Frostbite engine at Electronic Arts, where he contributed to *Battlefield*, *Battlefront*, *Mirror's Edge*, *Need for Speed*, and other games. Before that, he worked at Creative Assembly on the *Total War series* and *Alien: Isolation*.



Yaobin Ouyang is a developer technology engineer in NVIDIA. Currently his main job is researching RTX-based ray tracing techniques and providing consulting to game developers. He is enthusiastic about computer graphics and has investigated and implemented various rendering and physics simulation algorithms since he was in university. He also participated in the development of graphics engines in commercial products and successfully provided a voxel-based global illumination solution. He graduated from South China University of Technology in 2015, majoring in computer application technology.



Romain Pacanowski has been a research scientist at Inria Bordeaux-Sud-Ouest since December 2020. Before that, he was a research engineer at CNRS (French National Center for Scientific Research) working at the LP2N (Photonics, Numerical, and Nanosciences) laboratory. In 2009, he received his joint PhD from the Universities of Montreal and Bordeaux. After, he was a postdoctoral fellow at CEA (French Commission of Nuclear Energy) where he worked on BRDF modeling and measurement of retroreflection with a specialized gonio-spectrophotometer. In 2011, he was a postdoctoral fellow at BlackRock Studios, where he worked on real-time rendering of physically based materials. He has a strong background in light transport simulation and appearance modeling, but also in appearance measurement.



Georgios Papaioannou is an Associate Professor of the Department of Informatics at the Athens University of Economics and Business and head of the department's computer graphics group. His research focuses on real-time computer graphics algorithms, photorealistic rendering, illumination-driven scene optimization, and shape analysis, with more than 80 publications in these areas. He has participated as principal investigator, research fellow, and developer in many national, EU-funded, and industrial research and development projects and has worked in the past as a software engineer in virtual reality systems development and productions.



Daniel Parhizgar received his MS in computer science from KTH (Kungliga Tekniska Högskolan) Royal Institute of Technology. He did his master's thesis on hybrid refractions at Avalanche Studios and started working there as a graduate graphics programmer in February 2021. His interests in programming and game development came from the various Nintendo games he played growing up. During his free time, he also enjoys playing the guitar and learning about history.



Angelo Pesce currently leads the rendering efforts at Roblox. Previous to joining the metaverse, he was a Technical Director at Activision, where he helped the *Call of Duty* studios. His interest in computer graphics started in his teens by competing in the demoscene. In the past he has worked for companies such as Milestone, Electronic Arts, Capcom, and Relic. Angelo is actively involved in the computer graphics community, helping to organize conferences and presenting at venues including SIGGRAPH, GDC, and Digital Dragons. He a coauthor of the latest edition of *Real-Time Rendering* and sometimes writes on his *c0de517e* blog.



Matt Pettineo is the Lead Engine Programmer at Ready At Dawn Studios, where he heads the engineering team responsible for developing the in-house engine used for all of their games. He has worked at Ready At Dawn since 2009 and has contributed to *God of War Origins Collection*, *The Order: 1886*, *Lone Echo*, *Echo VR*, and *Lone Echo 2*. He often writes articles related to graphics programming on his blog, *The Danger Zone*, and is a coauthor of *Practical Rendering and Computation with Direct3D 11*.



Matt Pharr is a Distinguished Research Scientist at NVIDIA, where he works on ray tracing and real-time rendering. He is an author of the book *Physically Based Rendering*, for which he and the coauthors were awarded a Scientific and Technical Academy Award in 2014 for the book's impact on the film industry.



Matthew Rusch is a senior software engineer at NVIDIA in the Game Engines and Core Technology Group. Prior to joining NVIDIA, he spent the better part of two decades in the gaming industry working on graphics and game engines for various titles, spanning multiple console generations.



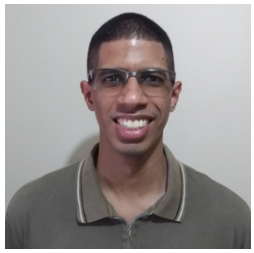
Manuele Sabbadin is a rendering researcher at Weta Digital, Wellington. He received his PhD in computer science from the University of Pisa, Italy, in 2019, where he also earned his MSc degree. For his PhD, he completed his research on techniques for the interaction and rendering of harvested 3D data at the Visual Computing Lab, ISTI(Institute of Information Science and Technologies), CNR (Italian National Research Council), Pisa. Since he joined Weta in 2019, he has worked, in large part, on the direct tracing of implicit surfaces.



Rodolfo Sabino is a PhD candidate in the Department of Computing at Universidade Federal do Ceará. His current work focuses on developing state-of-the-art parallel algorithms and structures for optimizing ray traversal in mesh-based scenes and enabling hardware-accelerated, physically based ray tracing. He is an experienced programmer in OpenGL and GLSL, developing his own hybrid stochastic path tracer, computer graphic, and linear algebra libraries. His research interests include hybrid rendering pipelines, stochastic numerical integration methods, and high-performance parallel graphics computing.



Peter Shirley is a distinguished research scientist at NVIDIA. He was formally a cofounder of two software companies and was a professor/researcher at Indiana University, Cornell University, and the University of Utah. He received a BS in physics from Reed College in 1985 and a PhD in computer science from the University of Illinois in 1991. He is the coauthor of several books on computer graphics and a variety of technical articles. His professional interests include interactive and high dynamic range imaging, computational photography, realistic rendering, statistical computing, visualization, and immersive environments.



Vinícius da Silva is a Postdoctoral Researcher at Pontifical Catholic University of Rio de Janeiro (PUC-Rio) and a collaborator at National Institute for Pure and Applied Mathematics' Vision and Graphics (VISGRAF) Laboratory. He obtained his DSc and MSc in computer science from the Federal University of Rio de Janeiro. His research interests are mostly related with real-time computer graphics, virtual reality, games, and machine learning, with a current focus on real-time ray tracing and its applications in procedural geometry, deep implicits, non-Euclidean spaces, and virtual reality. He also works with research and development in real-world problems involving data science and machine learning.



Juha Sjöholm joined the NVIDIA Developer Technology group in 2015 and has been working with cutting-edge rendering technologies in various games and engines. Before that, he worked for over a decade on the 3DMark benchmarks, ending as lead programmer. During the last few years, he has focused on the efficient application of RTX ray tracing in games, contributing to titles like *Control*, *Cyberpunk 2077*, and *Crysis Remastered*.



Josef Spjut is a Research Scientist at NVIDIA in the Graphics Research group. His work contributed to the ray tracing core hardware found in the Turing and Ampere GPUs. Lately, his research focus has shifted to esports and new human experiences. Prior to NVIDIA, he was a Visiting Assistant Professor in the Department of Engineering at Harvey Mudd College. He received a PhD from the University of Utah and a BS from the University of California, Riverside, both in computer engineering.



Nuno Subtil is a Rendering Engineer with Lightspeed Studios at NVIDIA, where he works on building real-time, fully ray traced renderers for game engines, with a particular focus on indirect illumination algorithms. With a background in physically based rendering, he has worked on topics ranging from engineering GPU graphics drivers to defining and evolving the latest graphics APIs, as well as contributing engineering expertise to several AAA game titles. He has also contributed to the field of bioinformatics, with research on GPU-based algorithms for processing DNA sequencing data as well as engineering work on a GPU-accelerated analysis pipeline for a nanopore-based DNA sequencing platform.



Marcus Svensson is a senior graphics programmer at Avalanche Studios, where he has worked on games such as *Rage 2* and *Just Cause 4*. He received his MS in game and software engineering from Blekinge Institute of Technology.



Kenzo ter Elst is a senior rendering engineer at Epic Games focusing on the low-level hardware interface of the Unreal Engine. Before that, Kenzo was a senior principal technical programmer at Guerrilla Games for eight years, maintaining, optimizing, and improving the rendering code of the Decima Engine. He has also been a Technical Director for Ryse at Crytek.



Will Usher is a Graphics Software Engineer at Intel, working on the oneAPI Rendering Toolkit. His work focuses on CPU and GPU ray tracing, distributed rendering, and scientific visualization. He completed his PhD in computer science in 2021 on data management and visualization for massive scientific data sets at the Scientific Computing and Imaging Institute at the University of Utah, advised by Valerio Pascucci.



Michael Vance is the Chief Technology Officer of Activision Publishing. He received his BSc in computer science in 1999 from Pennsylvania State University. His first job in the industry was as part of the start-up Loki Software, and he later began working at Treyarch before its acquisition by Activision. As a technical director, he has led engineering on the *Spider-Man* series and has contributed to many other titles, including all of the *Call of Duty* games from 2011's *Modern Warfare 3* to the present. He resides in Falmouth, ME, where he plays at farming when not volunteering for the Falmouth Land Trust.



Andreas A. Vasilakis is a postdoctoral fellow in the Department of Informatics of the Athens University of Economics and Business as well as an adjunct lecturer in the Department of Computer Science and Engineering at Ioannina University. His work has been published in high-ranking journals (CGF, TVCG, JCGT) and leading international conferences (EG, I3D, HPG), offering strong technical contributions to the fields of photorealistic rendering, geometry processing, and interactive graphics. Recently, Andreas cofounded Phasmatic, a company that pushes the boundaries of photorealistic 3D graphics on the Web.



Luiz Velho is a Senior Researcher and Full Professor at the National Institute for Pure and Applied Mathematics (IMPA) and leading scientist of the Vision and Graphics (VISGRAF) Laboratory. He received a BE in industrial design from Escola Superior de Disseny (ESDi), an MS in computer graphics from the Media Lab at the Massachusetts Institute of Technology (MIT), and a PhD in computer science from University of Toronto. He was a visiting researcher at the National Film Board of Canada, a Systems Engineer at Fantastic Animation Machine, and a Principal Engineer at Globo TV Network. He was also a visiting professor at the Courant Institute of Mathematical Sciences of New York University and a visiting scientist at the HP Laboratories and at Microsoft Research China.



Creto Augusto Vidal received his PhD in Civil Engineering from the University of Illinois at Urbana-Champaign in 1992. He is a professor of computer graphics in the Department of Computing at the Federal University of Ceará in Brazil. His current research interests are computer graphics, virtual reality applications, and computer animation. He has been the coordinator of several government-sponsored projects investigating the use of virtual reality and computer graphics.



Nick Vitsas is a PhD student in computer science at the Athens University of Economics and Business, where he also received his BSc and MSc. His current research focuses on photorealistic simulation and optimization of lighting conditions in 3D scenes. He is also the cofounder of Phasmatic, a company that aims to bring high-performance ray tracing to the Web.



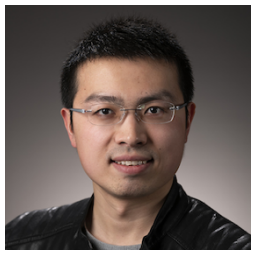
Ingo Wald is a director of ray tracing at NVIDIA. He received his master's degree from Kaiserslautern University and his PhD from Saarland University, then served as a postdoctorate at the Max-Planck Institute Saarbrücken, as a research professor at the University of Utah, and as technical lead for Intel's software-defined rendering activities (in particular, Embree and OSPRay). Ingo has coauthored more than 90 papers, multiple patents, and several widely used software projects around ray tracing.



Alan Wolfe is a graphics programmer and self-proclaimed Blue Noise Evangelist who is presently employed with NVIDIA and has two decades of professional game programming experience, originally self-teaching in the days of mode 13h, before hardware-accelerated graphics. Prior to NVIDIA, he was at Blizzard, working on various projects including *Starcraft 2*, *Heroes of the Storm*, *Diablo 4*, and a shared game engine team. He is the author of a programming blog with 250+ articles that covers a wide range of related topics including path tracing, neural networks, signal processing, skeletal animation, game physics, quantum computing, and mathematics.



Chris Wyman is a principal research scientist in NVIDIA's real-time rendering research group, where he explores problems including lighting, shadowing, global illumination, BRDFs, sampling, filtering, denoising, antialiasing, and building efficient data structures to solve these problems. Prior to NVIDIA, he received a PhD in computer science from the University of Utah, earned a BS from the University of Minnesota, and taught at the University of Iowa for nearly 10 years.



Ling-Qi Yan is an Assistant Professor of Computer Science at the University of California, Santa Barbara, co-director of the MIRAGE Lab, and affiliated faculty in the Four Eyes Lab. His research is in photorealistic rendering, including modeling visual appearance at real-world complexity, building theoretical foundations mathematically and physically to reveal the principles of the visual world, accelerating offline and real-time light transport, and exploiting lightweight machine learning approaches to aid physically based rendering. He is the recipient of the 2019 ACM SIGGRAPH Outstanding Doctoral Dissertation Award.



Xueqing Yang has more than 10 years of experience in game development. At the beginning, he entered TOSE Software, where he was sent to Japan for a half-year training in game developing; after that, he worked as a lead programmer and producer for many game titles. After he left TOSE, he joined Spicy Horse Software, working as a senior game programmer. He joined NVIDIA in 2011 and has been working as a developer technology engineer, focusing on creating advanced graphic techniques and helping the game developers use them to create high-quality visual effects in their games.



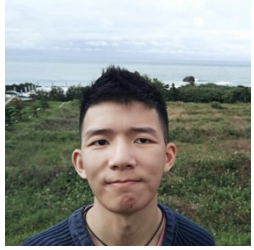
Cem Yuksel is an associate professor in the School of Computing at the University of Utah and the founder of Cyber Radiance LLC, a computer graphics software company. Previously, he was a postdoctoral fellow at Cornell University after receiving his PhD degree in Computer Science from Texas A&M University. His research interests are in computer graphics and related fields, including physically based simulations, realistic image synthesis, rendering techniques, global illumination, sampling, GPU algorithms, graphics hardware, modeling complex geometries, knitted structures, hair modeling, animation, and rendering.



Stefan Zellmann received a graduate degree in information systems from the University of Cologne and a doctorate degree in computer science in 2014, with a PhD thesis on high-performance computing and direct volume rendering. He has been the Chair of Computer Science at the University of Cologne since 2009. His research focuses on the interface between high-performance computing and real-time graphics. Stefan is primarily concerned with algorithms and software abstractions to leverage real-time ray tracing and photorealistic rendering at scale and for large 3D models and scientific data sets.



Zheng Zeng is a PhD student at the University of California, Santa Barbara, supervised by Professor Ling-Qi Yan, where he works on real-time ray tracing, offline rendering acceleration, and using neural networks to aid the physically based rendering process. He holds an MS in software engineering and a BS in digital media and technology from Shandong University.



Tianyi "Tanki" Zhang is a real-time rendering engineer at NVIDIA, working on the Omniverse RTX Renderer. Previously, he was a rendering engineer intern for the Unreal Engine at Epic Games, focusing on real-time ray tracing features. With his educational background in computer science, game development, and art design, his interests include real-time rendering algorithms, especially real-time ray tracing, and graphics systems.



Dmitry Zhdan is a senior developer technology engineer at NVIDIA, where he works with various game studios to bring best rendering techniques into modern games. This role also includes research and development, and the results of this work often push rendering in games to a new level.



Tobias Zirr received his master's degree in computer science from the Karlsruhe Institute of Technology, where he was also a PhD student in the Computer Graphics Group until 2021. He interned at the NVIDIA Real-Time Rendering Research group in 2015, dabbled in demoscene productions as a student, and also worked as a contractor developing shader creation tools for product visualization. His current research focuses on building a deeper understanding of robust Monte Carlo rendering algorithms and on new adaptations thereof for real-time rendering.

NOTATION

Here we summarize the mathematical notation used in this book. Vectors are denoted by bold lowercase letters, e.g., $\mathbf{v}$, and matrices by bold uppercase letters, e.g., $\mathbf{M}$. Scalars are lowercase, italicized letters, e.g., a and v. Points are uppercase, e.g., P. This is summarized in the following table:

Notation	What It Represents
P	Point
v	Scalar
$\mathbf{v}$	Vector
$\widehat{\mathbf{v}}$	Normalized vector
$\mathbf{M}$	Matrix

The components of a vector are accessed as

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} v_x & v_y & v_z \end{pmatrix}^{\mathsf{T}}, \tag{1}$$

where the latter shows the vector transposed, i.e., so a column becomes a row. To simplify the text, we sometimes also use $\mathbf{v} = (v_x, v_y, v_z)$, i.e., where the scalars are separated by commas, which indicates that it is a column vector shown transposed. We use column vectors by default, which means that matrix/vector multiplication is denoted $\mathbf{Mv}$. The components of a matrix are accessed as

$$\mathbf{M} = \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{m}_0, \mathbf{m}_1, \mathbf{m}_2 \end{pmatrix}, \tag{2}$$

where the $\mathbf{m}_i$, $i \in \{0, 1, 2\}$, are the column vectors of the matrix. For normalized vectors, we use the following shorthand notation:

$$\widehat{\mathbf{d}} = \frac{\mathbf{d}}{\|\mathbf{d}\|}, \tag{3}$$

i.e., if there is a hat over the vector, it is normalized. A transposed vector and matrix are denoted $\mathbf{v}^\mathsf{T}$ and $\mathbf{M}^\mathsf{T}$, respectively.

A direction vector on a sphere is often denoted by ω and the entire set of directions on a (hemi)sphere is Ω. Finally, note that the cross product between two vectors is written as $\mathbf{a} \times \mathbf{b}$ and their dot product is $\mathbf{a} \cdot \mathbf{b}$.





PART I
RAY TRACING FOUNDATIONS

PART I

RAY TRACING FOUNDATIONS

This part covers various fundamental topics in ray tracing, including camera ray generation, bounding box construction, ray intersection, soft shadow edges, texture filtering, ideal reflection and refraction, motion blur, and translucency. We start with a summary of photographic terms as they apply to ray tracing and end with a full implementation of a reference path tracer. Along the way, we also explore a topic in spectral rendering of volumes. Many of the chapters in this part include source code snippets to make the techniques simple to understand and use.

Chapter 1, *A Breakneck Summary of Photographic Terms (and Their Utility to Ray Tracing)*, explains terminology used in photography and videography in terms understandable to ray tracing authors and users. Photography and videography terminology has developed over more than a century, and some of it is rather confusing, or downright misleading. This gem explains it all!

One of the most commonly used operations in any efficient ray tracer is to determine whether a ray hits an axis-aligned bounding box (AABB). Chapter 2, *Ray Axis-Aligned Bounding Box Intersection*, describes the most optimal implementation of a ray/AABB intersection test, including some neat tricks.

To generate an image, a ray tracer shoots many rays from the camera position. Chapter 3, *Essential Ray Generation Shaders*, describes the most common projections and lens approximations, and how to map image (pixel) positions to ray directions.

When object geometry is represented with coarse triangles or quadrilaterals, each covering many pixels, a problem often arises: the edge between illuminated and shadowed parts of the object is very sharp—much sharper than the otherwise smooth shading. This is called the *shadow terminator problem*. Chapter 4, *Hacking the Shadow Terminator*, describes a method that moves the shading points to overcome this pesky issue.

When sampling textures, it is important to access the appropriate texture resolution (mipmap level) to achieve an optimal balance between texture blur and sampling noise, while maximizing cache performance. Several gems

cover various aspects of this topic. The first is Chapter 5, *Sampling Textures with Missing Derivatives*. It is common to use texture coordinate differentials when sampling textures. This chapter describes what to do if such differentials are not available, introducing an efficient approach based on the camera matrix and geometry visible in a pixel.

Ray cones are a fast, approximate way to calculate the mip level for texture lookups. Chapter 6, *Differential Barycentric Coordinates*, describes a simple and efficient method to compute differentials of barycentric coordinates, which in turn can provide the differentials of texture coordinates needed for texture filtering. Chapter 7, *Texture Coordinate Gradients Estimation for Ray Cones* reformulates the problem to utilize hardware-supported instructions. This maintains almost the same visual results and leads to more convenient shader code.

Mirror reflection and ideal refraction is at the heart of classic recursive ray tracing. Chapter 8, *Reflection and Refraction Formulas*, provides a clear and concise refresher on how to compute reflection and refraction directions.

The magnitude of reflection and refraction follows the Fresnel equations, which can be approximated with Schlick's simple formula. Chapter 9, *The Schlick Fresnel Approximation*, discusses the accuracy of this approximation for dielectric materials and presents an extended Schlick approximation for metals.

Some of the chapters earlier in this part describe how to use ray cones for texture map sampling and how to propagate ray cones at reflections. Chapter 10, *Refraction Ray Cones for Texture Level of Detail*, extends this with a recipe for propagating ray cones at *refractions*.

Chapter 11, *Handling Translucency with Real-Time Ray Tracing*, describes various practical approaches to efficiently render translucent materials in a real-time ray tracing setting, for example reusing shading results computed for directly visible points.

When rendering moving geometry, the bounding volume hierarchy needs to represent the motion. Chapter 12, *Motion Blur Corner Cases*, describes how to combine geometry from different sample times, how to bound geometry that has both transformation and deformation motion, and how to deal with incoherent motion.

For spectral rendering of volumes, one usually needs to convert RGB inputs for the volume attenuation coefficients to spectral values. Chapter 13, *Fast Spectral Upsampling of Volume Attenuation Coefficients*, introduces a simple and fast method to do this conversion that gives very good results.

Chapter 14, *The Reference Path Tracer*, describes a simple path tracer that serves two purposes: (1) to generate reference images to compare other implementations against and (2) to supply code examples that can be used as a foundation for adding path tracing to existing rendering engines.

The information in this part builds upon and supplements the basics of modern ray tracing introduced in 2019's *Ray Tracing Gems*. It is intended to help you better understand the fundamentals of modern ray tracing. Enjoy!

Per Christensen

CHAPTER 1

A BREAKNECK SUMMARY OF PHOTOGRAPHIC TERMS (AND THEIR UTILITY TO RAY TRACING)

Trevor David Black
Google

ABSTRACT

The pursuit of photography and videography has a dizzying number of terms to remember. It takes months of practice and thousands of shots to understand at an intuitive level how photographers talk about their craft. As quickly as possible, this chapter aims to explain the terms of photography and videography—boiling them down to their most basic mathematics and utility—before quickly covering their significance to ray tracing and outlining feasible implementations for a select few terms.

1.1 INTRODUCTION

Photography is a hobby and a profession that is absolutely filled with misinformation. This should hardly come as a surprise: The fundamental aspects of photography are deeply rooted in optical physics and electromagnetism. Getting started with photography is already difficult enough—there are so many terms and fiddly details—that expecting an amateur to also develop a keen understanding of undergraduate physics is simply asking for too much. This problem is compounded by the amount of information that is created by photographers to simplify these concepts for photographers, and the information that is created to simplify those simplifications. It would be remarkable for anyone to develop a keen understanding of the grammar of the hobby and of the underlying science without digging through academic texts. If you're reading this, then it is assumed that you are interested in learning both how photography works as a craft and how photography works from a fundamental mathematics and physics perspective. It is also assumed that you're interested in taking concepts learned from photography and adapting them for use within a ray tracing context. You will find reading this chapter that a ray tracing camera is

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_1

in many respects an idealized camera, and true—physical—camera systems leave a lot to be desired. That said, you'll frequently encounter cases where the idiosyncrasies of physical systems need to be modeled within virtual contexts, the big examples being depth of field, chromatic aberration, and rolling shutter. If you work at a studio, this understanding might allow you to have more in-depth conversations with the cinematographers and camera personnel on set. Starting with the basics, this chapter provides a concise and utilitarian summary of the most common terms of photography.

PHOTOGRAPHY A hobby and/or profession infected with the obsession of expressing ideas through the capturing of photons.

CAMERA A device that captures photons and embeds them in a storage medium for later viewing.

1.2 DIGITAL SENSOR TECHNOLOGY

PHOTODETECTOR A passive electronic element that converts photons into electrical current. Photodetectors usually have a nonlinear response curve. The same as a photosensitive diode.

PHOTOSITE A single photodetector. A single pixel.

SENSOR A plane of photosites aligned to a Cartesian grid. The sensor is responsible for sampling and digitizing the electrical currents produced by incident photons for each individual photocell.

SENSOR MODULE The electrical module that contains the digital sensor and all associated optical filters.

Every sensor module has a unique transfer function. This can depend on a tremendous number of factors and is usually not worth modeling in its entirety. Rather, if you need to simulate a specific sensor module, it is usually best to sample that sensor directly. For sampling digital sensors, see Physlight [8]. For an example of a fictional spectral sensitivity graph, see Figure 1-1.

PIXEL PITCH The physical distance between the centers of any two adjacent pixels, usually measured in micrometers.

IR/UV FILTER An optical filter placed in front of the sensor to absorb infrared (IR) and ultraviolet (UV) wavelengths. The sensitivity of the photodetectors can vary wildly from one product to the next, but the

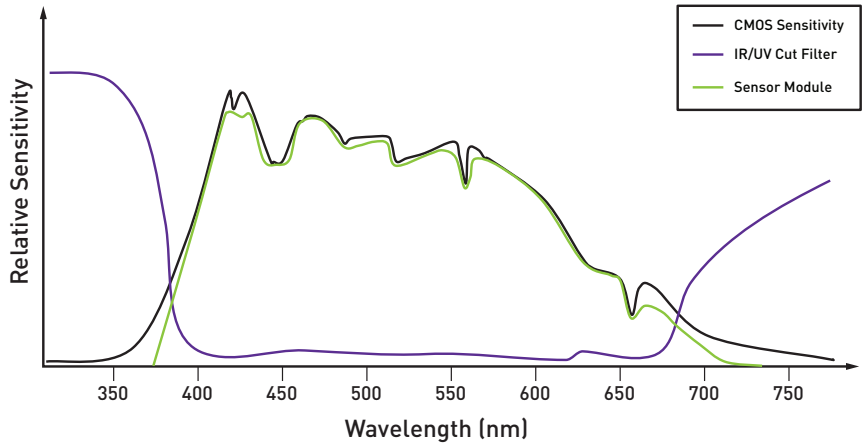


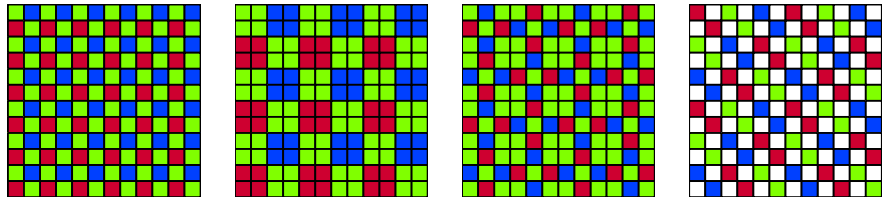
Figure 1-1. *This figure depicts three separate things. First, it shows a fictional spectral sensitivity (black) that extends outside of the normal perception of the electromagnetic spectrum. To compensate for this unnecessary extension, an IR/UV cut filter (purple) is depicted that blocks IR and UV photons. Finally, the combination of the two is depicted as the final spectral sensitivity of the sensor module (green) that more closely matches human perception.*

photodetectors found in most cameras are sensitive to wavelengths outside of human sensation. Specifically, they can go as short as 300 nm (ultraviolet) to as long as 1200 nm (infrared). The IR/UV optical filter allows a sensor to more closely approximate human sensitivity.

IR and UV should not be a problem for you unless you are working with assets that emit those frequencies. If you are attempting to emulate the photodetectors of a specific sensor, failure to include the IR/UV filter will produce an inaccurate, washed-out image.

BLACK AND WHITE SENSOR MODULE A grid of photosites with an IR/UV filter. Each photosite captures all incident photons with wavelengths between—roughly—400 nm and 700 nm. The sensor outputs a single numerical value for each pixel that represents the integral of all photons over the capture period. A black and white sensor captures luma (brightness) data only and has no capacity for chroma (color) data. A fictional sensor module with its IR/UV cut filter and final spectral sensitivity can be seen in Figure 1-1.

COLOR SENSOR A black and white sensor with an additional optical filter that is made up of a mosaic of colored glass. The mosaic has a repeating pattern that specifies a colored optical filter for each individual pixel. This



(a) *Bayer pattern* **(b)** *Quad Bayer* **(c)** *X-Trans* **(d)** *URSA 12K*

Figure 1-2. *A depiction of three common sensor mosaics (Bayer, Quad Bayer, and X-Trans) and one uncommon sensor mosaic (URSA Mini Pro 12K).*

colored glass is typically red, green, blue, or clear. A red filter will approximate the human long cone, a green filter will approximate the human medium cone, and a blue filter will approximate the human short cone. The LMS (long, medium, short) system is defined in *Colorimetry* [9]. One can crudely approximate the degradation of chroma and luma energy for any given mosaic pattern by converting from RGB to XYZ [3].

BAYER PATTERN The most common mosaic pattern found in color sensors [1]. Others do exist and are used, see Sony [10], Fujifilm [4], and Buettner [2]. A 2×2 repeating pattern consisting of two green filters, one red filter, and one blue filter.

The two green filters are along one diagonal, the red and blue filters along another. The mosaic for the Bayer filter is depicted in Figure 1-2, alongside other mosaic patterns of note. The Bayer filter captures roughly 45% of the luma data and 29% of the chroma data. The information capture of the Bayer filter is crudely approximated as follows:

$$\begin{bmatrix} 0.49000 & 0.31000 & 0.20000 \\ 0.17697 & 0.81240 & 0.01063 \\ 0.00000 & 0.01000 & 0.99000 \end{bmatrix} \cdot \frac{1}{4} \left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + 2 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 0.3275 \\ 0.4531 \\ 0.2525 \end{bmatrix} \tag{1.1}$$

DEMOSAICING The signal processing algorithm that approximates full RGB color for each pixel. This will be different for each mosaic pattern. Before demosaicing, each pixel in the sensor will only have a single color or luma value. Full RGB displays require three data points for each pixel.

Knowing the specific demosaicing algorithm may be necessary if compositing on top of a raw recording.

FOVEON SENSOR A sensor made up of three vertically stacked photodiodes in the bulk of the semiconductor that removes the need for a mosaic pattern. Each photocell consists of a stack of three photodiodes one each for red, green, and blue wavelengths. Each photocell reads out a full RGB triple, allowing for full RGB images without the need for demosaicing. The creation of the Foveon sensor was chronicled in *The Silicon Eye* [5].

A useful mental model is treating ray tracing cameras as being similar to Foveon sensors.

ALIASING A distortion or artifact that causes a sampled signal to be different from the original signal. This is common in images with places of high-frequency data. A good place to find aliasing in photographs is in fabric. Aliasing can be especially problematic in sensor modules with mosaics as adjacent photocells can have different filters and capture different information. In Bayer patterns this can lead to high-frequency noise where green and red information alternate (or green and blue alternate).

This is the same type of aliasing that is seen in ray tracing. Aliasing in physical capture tends to be more distracting due to mosaics.

ANTIALIASING FILTER An optical filter placed in front of a sensor to reduce the presence of aliasing artifacts in an image. It is effectively an optical blurring element that reduces high-frequency information from reaching the sensor. Special attention is given to removing moire, as it is the most distracting of aliasing artifacts.

MOIRE A distracting visual artifact that appears when two patterns are imperfectly overlapped. Patterns of high-frequency detail overlap and form visually striking low-frequency patterns.

1.3 FILM

FILM A physical capture and storage medium, often a photosensitive sandwich of chemicals layered onto a plastic substrate. The film is placed at the rear of the camera to capture incoming photons. Depending on the type of camera, film stock can be loaded into the camera one at a time or can be wound through the camera in either the horizontal or vertical direction.

FILM NEGATIVE A storage medium for film, also the raw capture out of the camera. For a film stock with the negative characteristic, increasing photons darkens the image. The output from a film negative will have inverted luma and chroma data.

FILM POSITIVE (REVERSAL FILM) A film stock with the positive characteristic. The output from the film does not have inverted luma and chroma data.

NEGATIVE A negative image. Colloquially used to refer to any immediate output from a capture device.

35MM FILM A film stock where the width of the roll is 35 mm. This is the most commonly used film throughout most of the twentieth century.

8MM FILM A film stock where the width of the roll is 8 mm.

16MM FILM A film stock where the width of the roll is 16 mm.

70MM FILM A film stock where the width of the roll is 70 mm.

65MM FILM A film stock where the width of the roll is 70 mm. 70mm film is sometimes referred to as 65mm for historical reasons. Note that the photosensitive region of "65mm" film is not 65 mm in any dimension.

1.4 COMMON CAPTURE DIMENSIONS

ASPECT RATIO The ratio of the horizontal dimension to the vertical dimension of the photosensitive region, represented in an $X : Y$ format. For film stock this is the ratio of the physical dimensions of the photosensitive region. For digital sensors this is the ratio of the horizontal pixel count to the vertical pixel count.

35MM PHOTOGRAPHIC FILM The film stock most common to photographic applications in the twentieth century. The film traverses through the camera horizontally, such that the film is unwound with the perforations at the top and bottom of the camera. The 35mm photographic film stock has a photosensitive region with a vertical resolution of 24 mm. This is due to the space taken up by the perforations above and below the photosensitive region. 35mm photographic film has a horizontal resolution of 36 mm. Leading to a complete resolution of 36 mm by 24 mm and an aspect ratio of 3:2.

35MM MOVIE FILM The film stock most common to moviemaking in the twentieth century. The film traverses through the camera vertically, such that the film is unwound with the perforations at the sides of the camera. There is an audio strip that stores captured audio that sits between the perforations and the photosensitive region on one side of the film stock. 35mm movie film

has a horizontal resolution of 22 mm and a vertical resolution of 16 mm, leading to a complete resolution of 22 mm by 16 mm and an aspect ratio of 1.375:1.

When creating a ray tracing sensor designed to mimic a real film camera, don't just use the width of the film stock as the width of your virtual sensor, i.e., don't just set your width to 8 mm, 16 mm, 35 mm, or 70 mm. The real photosensitive region is never the full width as there may be an audio strip or perforations.

35MM SENSOR (FULL FRAME) A digital sensor with a photosensitive region designed to mimic 35mm photographic film. It has arbitrary pixel resolution with a sensor size of 36 mm by 24 mm and an aspect ratio of 3:2.

CROPPED SENSOR (APS-C) A digital sensor with a photosensitive region that has roughly half the area of 35mm photographic film. It has arbitrary pixel resolution with a sensor size approximating 23.4 mm by 15.6 mm and an aspect ratio of 3:2.

FOUR THIRDS SENSOR A digital sensor with a photosensitive region that has roughly one quarter the area of 35mm photographic film. It has arbitrary pixel resolution with a sensor size approximating 17.3 mm by 13 mm and an aspect ratio of 4:3. The *Four Thirds* name does not come from the aspect ratio, rather it refers to the associated cathode ray tube size.

SUPER 35MM SENSOR A digital sensor with a photosensitive region designed to mimic 35mm movie film. It has arbitrary pixel resolution with a sensor size of 24.89 mm by 18.66 mm and an aspect ratio of 1.33:1.

MEDIUM FORMAT Photography taken using a photosensitive region (film or sensor) that is larger than 35mm photographic film but smaller than large format (photography). This means that medium format photosensitive regions fall between 36 mm by 24 mm and 130 mm by 100 mm. Medium format can also be of arbitrary aspect ratio.

LARGE FORMAT (PHOTOGRAPHY) Photography taken using a photosensitive region (film or sensor) that is 130 mm by 100 mm or larger.

LARGE FORMAT (VIDEOGRAPHY) Videography taken using a photosensitive region that is larger than a Super 35mm sensor.

1.5 COMMON CAPTURE RESOLUTIONS

2K (DCI 2K) A pixel resolution of 2048 by 1080.

4K (DCI 4K) A pixel resolution of 4096 by 2160.

DCI 8K A pixel resolution of 8192 by 4320.

HIGH-DEFINITION (HD) A pixel resolution of 1920 by 1080.

ULTRA-HIGH-DEFINITION (UHD) A pixel resolution of 3840 by 2160.

8K A pixel resolution of 7680 by 4320.

1.6 LENSING

LENSING A mathematical operation that bends incoming parallel lines such that they converge toward a single central point. Lensing is depicted in Figure 1-3 with a simple thin lens model.

The thin lens model is used frequently in ray tracing cameras and may satisfy the majority of your visual needs.

FOCAL POINT The single central point toward which parallel lines are bent.

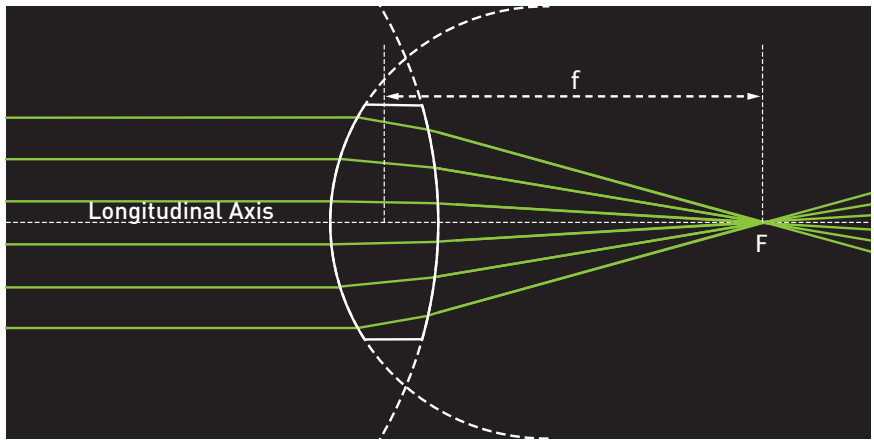


Figure 1-3. *The lensing of incoming photons by a thin lens model. The focal length is determined by the intersection of the incoming parallel lines and the lensed lines. Both faces of the optical element model the surface of a sphere. The focal point is denoted with F. The focal length is denoted with f.*

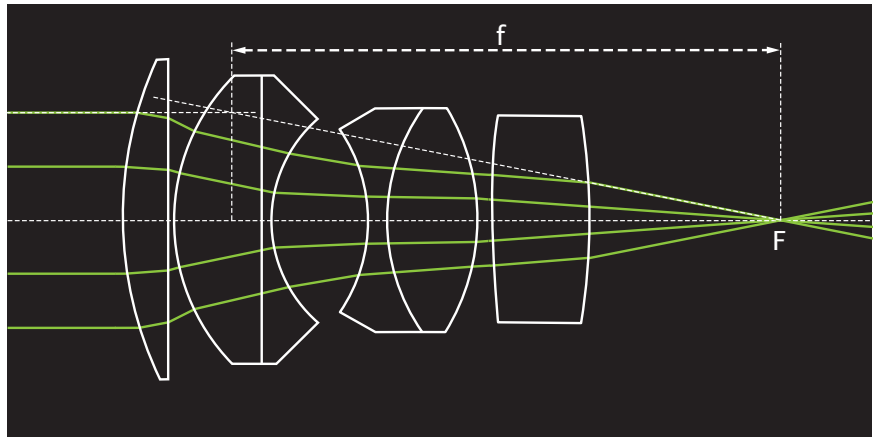


Figure 1-4. *The wonderful Carl Zeiss Jena Biotar 58/2 photographic lens. Real photographic lenses can have incredibly complicated optical and mechanical designs. The lensing drawn here is a slightly wrong exaggeration to more clearly depict the focal length.*

PHOTOGRAPHIC LENS A physical enclosure made up of optical elements that lenses incoming photons toward a focal point in front of the photosensitive region. A fictional photographic lens is depicted in Figure 1-4.

FIELD OF VIEW The amount of the universe in front of the lens that is visible to the focal point. This is represented by a solid angle and is usually measured in degrees or radians for each dimension.

HORIZONTAL FIELD OF VIEW The angle, in radians or degrees, that constitutes the widest view of the universe visible to the focal point.

VERTICAL FIELD OF VIEW The angle, in radians or degrees, that constitutes the tallest view of the universe visible to the focal point.

PINHOLE CAMERA A blackbody box with a small hole punched in one side and a photosensitive region on the opposite side. The small hole, called an aperture, is the focal point for a pinhole camera. An ideal pinhole camera has minimal defocus blur, but lets in very few photons.

FOCAL LENGTH The distance from the focal point to the plane where the incident parallel lines and the converging lines intersect. This is a physical property of the lens, and the camera has no effect on it. The focal length is shown in Figures 1-3 and 1-4. The horizontal and vertical dimensions of the field of view can be calculated from the focal length (and vice versa) with the

following:

$$h_{\text{fov}} = 2 \arctan\left(\frac{\text{photosensitive width}}{2 \cdot \text{focal length}}\right), \tag{1.2}$$

$$v_{\text{fov}} = 2 \arctan\left(\frac{\text{photosensitive height}}{2 \cdot \text{focal length}}\right). \tag{1.3}$$

APERTURE/IRIS An opening in the camera and lens that is (ideally) the sole source of photons to the photosensitive region. The apertures of interchangeable lenses are usually of variable size where the radius can expand or contract. An increase in radius lets in more photons but increases the size of the circle of confusion. A decrease in radius lets in fewer photons but decreases the size of the circle of confusion. The term *aperture* is commonly used in photography, whereas the term *iris* is more commonly used in videography, but they both represent the same thing.

CIRCLE OF CONFUSION An optical blur caused by the convolution of photon cones emanating from adjacent scene features. The radii of these photon cones are directly proportional to the aperture/iris.

FOCAL PLANE (PLANE OF FOCUS) An idealized plane orthogonal to the longitudinal (forward) axis of the lens. Any object or feature intersecting the plane of focus is projected onto the sensor plane in the maximum detail that is possible for that lens.

DEFOCUS BLUR A blurring caused by the circle of confusion for any object not intersecting the focal plane.

DEPTH OF FIELD The distance between the two planes, one near and one far, wherein all objects appear sharp or in focus. All objects and features that do not perfectly intersect the plane of focus will experience defocus blur. A crude simplification is to say that a feature will only appear sharp if the circle of confusion is smaller than the pixel pitch of the sensor. Depth of field can sometimes colloquially be used to refer to defocus blur. This is wrong. Don't do this.

STOP A doubling. Multiple stops represent powers of two. One stop is 2×, two stops is 4×, three stops is 8×, etc.

ENTRANCE PUPIL The appearance of the aperture as viewed from the front of the lens. The diameter of the entrance pupil is not the diameter of the aperture. The entrance pupil can be thought of as the aperture as "seen" by the photons incident to the front lens element.

APERTURE NUMBER The ratio of the focal length to the diameter of the entrance pupil. Also called the *f*-number, it is a physical property of the lens. Note that as the aperture width decreases, the aperture number goes up (as the entrance pupil is correlated with aperture width). The *f*-number is not to be confused with *f*, which denotes the focal length in the figures.

F-STOP The quantization of the aperture number. The f-stop is usually written in terms of powers of $\sqrt{2}$ (e.g., f1.4, f2.0, f2.8, etc.), which are one stop apart, or powers of $\sqrt{2/3}$ (e.g., f1.4, f1.6, f1.8, f2.0, etc.), which are one third of a stop apart. As an example, a lens with an f-stop of f2.0 has a focal length that is twice the width of its entrance pupil.

TRANSMITTANCE The light transmission efficiency of a lens. If 100% of the incident photons to the front element of a lens are transmitted to the rear of a lens, the lens has a transmittance of 1.0. The successive layers of glass in a lens can cause some of the photons to be absorbed in the lens wall or reflected back out into the scene. It is common to see lenses with a transmittance between 0.7 and 0.9.

T-STOP The f-stop of a lens modified by the transmittance of the lens. The t-stop of a lens is calculated as

$$T = \frac{\text{f-stop}}{\sqrt{\text{transmittance}}}, \tag{1.4}$$

where the f-stop is a numerical value (e.g., 1.4, 2.0, 2.8). As an example, an f2.0 lens with a transmittance of 0.7 and an f2.4 lens with a transmittance of 1.0 will have a similar quantity of photons transmitted through to the sensor.

1.7 SHUTTER

SHUTTER SPEED The duration that the photosensitive region is sensitive to photons.

FRAME RATE The number of photographic frames that are captured per second for a video. Common frame rates are 24 frames per second and 23.976 (24/1.001) frames per second for movies in NTSC countries, 30 frames per second and 29.97 (30/1.001) frames per second for television in NTSC countries, and 25 frames per second for both movies and television in PAL countries. The frame rate must be lower than the inverse of the shutter speed.

SHUTTER ANGLE The duration that the photosensitive region is sensitive to photons normalized by the inverse of the frame rate. A shutter speed of zero is 0 degrees. A shutter speed of one over the frame rate is 360 degrees:

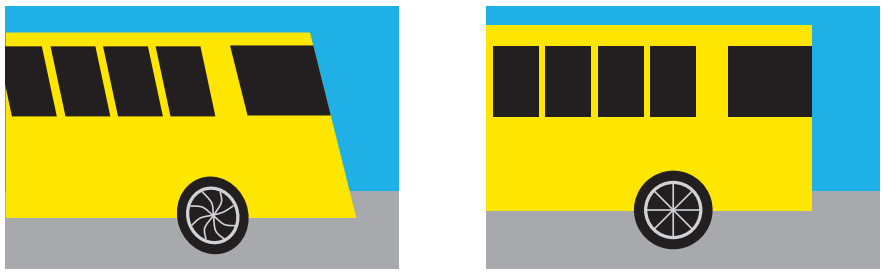
$$\text{shutter angle} = \text{shutter speed} \cdot \text{frame rate} \cdot 360 \text{ degrees} \tag{1.5}$$

A commonly seen example in movies is a shutter time of 20 ms for 24 frames per second, which has a shutter angle of 172.8 degrees.

180 DEGREE RULE The appearance of smooth and lifelike motion blur in videographic work is considered to be the strongest when shooting with a shutter angle of 180 degrees. For films shot in 24 frames per second, the shutter speed would be set to 1/48 seconds, or 21 ms.

ROLLING SHUTTER An undesirable delay in the readout of pixel values causing top left pixels to be sampled before bottom right pixels. If every pixel has a 1 nanosecond delay after the previous pixel, a 4K readout will delay 8,000,000 ns, or 8 ms. If you are shooting at 30 frames per second, corresponding to a 33 ms frame time, then the very last pixel will be offset from the first pixel by nearly a quarter of a frame. If you are compositing work onto real camera sensors, you may need to add rolling shutter to your render. Rolling shutter is depicted in Figure 1-5a.

GLOBAL SHUTTER All pixel values are sampled at the same time. A global shutter has no rolling shutter. Global shutter is depicted in Figure 1-5b.



(a) *Rolling shutter*

(b) *Global shutter*

Figure 1-5. *An image depicted with rolling shutter and without rolling shutter. Notice that moving objects will distort as parts of the frame are captured later. Take special notice that rotating objects, such as wheels and rotors, will take on an unnatural looking S-curve shape.*

1.8 EXPOSURE

EXPOSURE A quantization of the amount of light captured in an image. The exposure at the focal plane is fully described by

$$\text{exposure} = \frac{\mathrm{L}t\pi(U-f)^2 TCh\cos^4(\theta)}{4A^2U^2}, \tag{1.6}$$

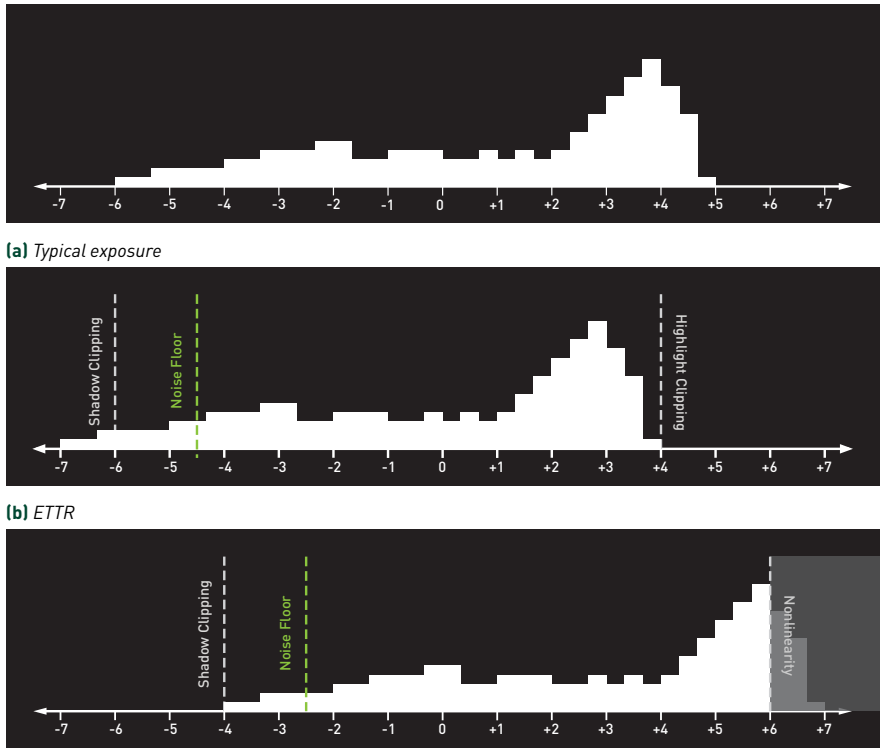
where L is the scene luminance, t is the shutter time, U is the distance to the focal plane, f is the lens's focal length, A is the lens's f-number, T is the transmittance of the lens, C is the camera flare correction factor, h is the vignetting factor, and θ is the angle of the image point from the longitudinal axis of the lens. The mathematics is condensed from ISO 12232 [6].

ISO/GAIN Digital amplification. This is a gross oversimplification. It is recommended that you speak to your asset creator about how they are most comfortable specifying asset exposure. This is most commonly done in terms of (1) ISO, (2) exposure value (EV), (3) decibels, or (4) f-stop. The ISO of a digital sensor is thoroughly defined in ISO 12232 [6]. The term *ISO* is commonly used in photography, whereas the term *gain* is more commonly used in videography, but they both represent the same thing. Even though ISO is an acronym (referring to the International Organization for Standards), it is pronounced "iso" as in the Greek prefix meaning "equal."

CLIPPING The brightness beyond which a photosensitive region is no longer able to distinguish differences in exposure. Digital sensors clip in the highlights, so there is a maximum brightness above which any additional light will only be read out as the clipping value. Negative film clips in the shadows, so there is a minimum brightness below which any reduction in light will only be read out as the clipping value.

Clipping can occur in digital sensors due to either the saturation of photosites or the maximum representable value in a fixed point number system being surpassed. Simple ray tracers are commonly written such that they clip at the white point for the intended final media format.

NOISE FLOOR For a digital sensor, the exposure value at which the signal-to-noise ratio falls below a certain value (this value is usually 2:1 or 1:1). The primary sources of noise in captured images are photon shot noise, dark current noise, analogue processing readout noise of image sensors, and quantization noise of A/D (analog to digital) converters. Noise characteristics of digital cameras is defined in ISO 15739 [7].



(a) *Typical exposure*

(b) *ETTR*

(c) *ETTL*

Figure 1-6. *A typical histogram for a scene with a single bright source of light. A value of 0 represents middle gray. Each mark is a stop up or down. Digital sensors have a much smaller highlight latitude than film; in this example, digital sensors clip only at 4 stops. Conversely, digital sensors have much higher shadow latitude, 6 stops versus film's 4 stops. Exposing to the right in a digital camera pushes as much of the scene out of the noise floor as possible. Exposing to the left for film keeps as much of the scene out of the nonlinearity as possible.*

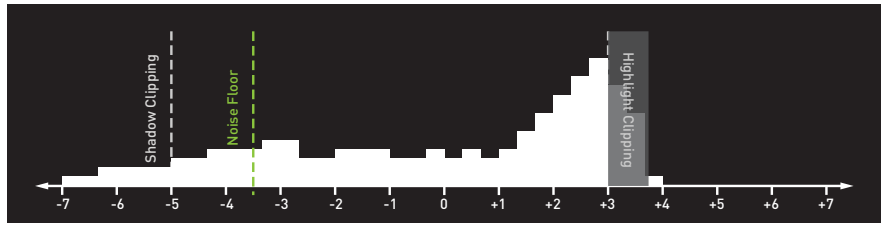
DYNAMIC RANGE For digital sensors, the ratio of the brightness at clipping and the brightness of the noise floor of the sensor. Dynamic range is depicted along with ETTR and ETTL in Figure 1-6.

EXPOSURE TO THE RIGHT (ETTR) Photos taken with a digital sensor should be taken with the settings that produce the highest exposure without clipping the highlights. This maximizes detail in both the shadows and the highlights. If a histogram is a left-right axis with shadows on the left and highlights on the right, the digital photographer is encouraged to push their exposure to the right.

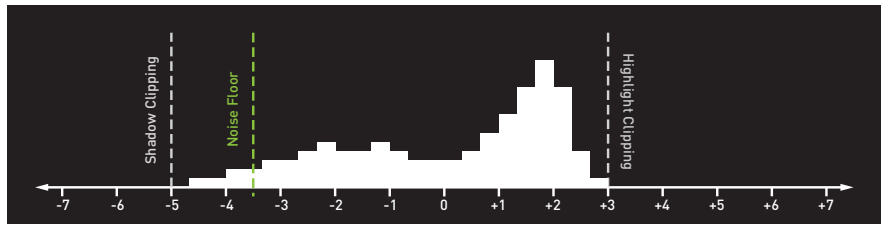
EXPOSURE TO THE LEFT (ETTL) Photos taken with a negative film should be taken with the settings that produce the lowest exposure without clipping the shadows. This maximizes detail in both the shadows and the highlights. If a histogram is a left-right axis with shadows on the left and highlights on the right, the film photographer is encouraged to push their exposure to the left.

WHITE BALANCE The color representation of white. Both the camera and the lights in a scene will have specific white balances. These may be tunable by color temperature along with various hue and/or saturation adjustments.

LOG RECORDING An increase in the dynamic range of video recording by moving the luma gamma from a linear to a logarithmic curve. For cameras that record video luma in 8 or 10 bits, the predominant source of noise is quantization noise from the A/D converters. Compressing the luma curve to fit within the decreased bit count with a log curve helps reduce quantization errors. The video can then be reversed in the edit to capture maximal dynamic range. Recording in log may complicate exposing to the right due to the compressed information in the highlights. The dynamic range boosting effects of log recording can be seen in Figure 1-7.



(a) *Linear gamma*



(b) *Logarithmic gamma*

Figure 1-7. *The same histogram data that is presented in Figure 1-6, only now reduced to an 8-bit A/D converter. The drop in dynamic range guarantees that some part of the scene will either clip (as a highlight) or appear with an objectionable amount of noise. Compressing the scene using a log gamma allows a much better portrayal of the scene, and allows us to protect our highlights without too much noise.*

RAW RECORDING A special type of video capture that enables additional modifications in the edit. What exactly raw capture entails depends on the camera manufacturer and on the raw protocol. The raw capture may support adjusting ISO or white balance while editing, may already be demosaiced, may be compressed, or may be compressed with lossy compression.

1.9 EQUIVALENCY

BLOCKING The placement of objects and people within the image. The look and feel of an image can be finely tuned by varying the position of subjects and other contrasting elements within the frame.

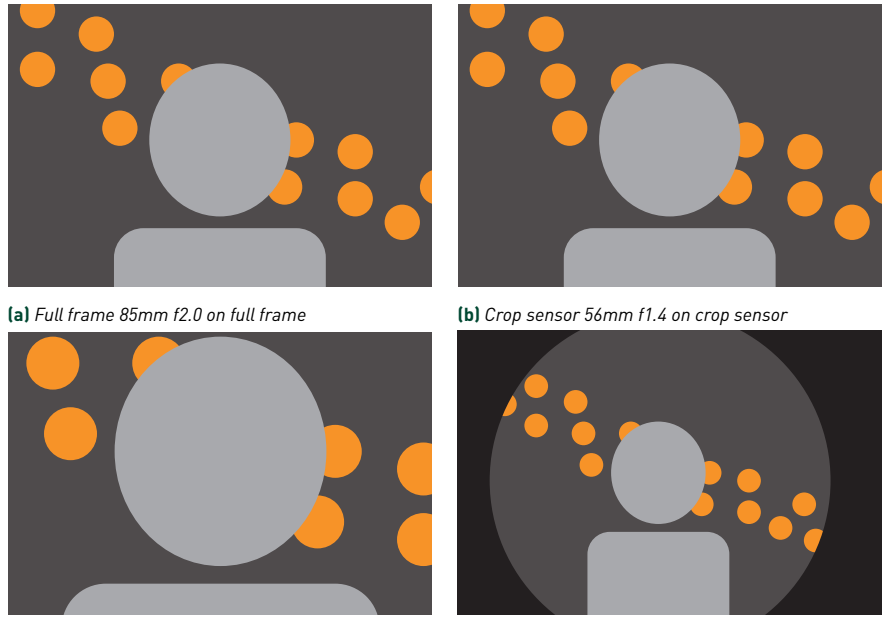
35MM EQUIVALENT A standardized framework by which a photographer might reason about their camera or lens. A lens that is designed for a camera without a 35mm photographic sensor may be described in terms of a similar lens that is designed for a 35mm photographic sensor. In this way a photographer can reason about the field of view and the depth of field for a lens provided they've already memorized these qualities for a lens of the 35mm photographic sensor. This removes the need to memorize field of view and depth of field for all sensor sizes, as they can be computed for any sensor size quickly.

CROP FACTOR A tool used to compare the photometric qualities of cameras with differing photosensitive region sizes. The crop factor can be either:

1. The ratio of the film/sensor width to the width of 35mm photographic film:
$$\text{crop factor} = \frac{\text{width}}{36\text{mm}}. \tag{1.7}$$
2. The ratio of the film/sensor diagonal to the diagonal of 35mm photographic film:
$$\text{crop factor} = \frac{\text{diagonal}}{43\text{mm}}. \tag{1.8}$$

35MM EQUIVALENT FOCAL LENGTH The physical focal length of the lens multiplied by the crop factor of the photosensitive region. 35mm equivalent focal length and 35mm equivalent depth of field are depicted in Figure 1-8.

35MM EQUIVALENT EXPOSURE The exposure of a lens does not vary with crop factor. The sensitivity and noise floor of the camera might change with crop factor (owing to the smaller/larger photosensitive region), but two different cameras set to the same ISO/gain value will capture the same scene with similar exposure values.



(a) *Full frame 85mm f2.0 on full frame*

(b) *Crop sensor 56mm f1.4 on crop sensor*

(c) *Full frame 85mm f2.0 on crop sensor*

(d) *Crop sensor 56mm f1.4 on full frame*

Figure 1-8. *A depiction of the impact of sensor size on field of view. If you start with a static tripod, a static subject, and a static background of point lights, your choice of focal length and sensor size has a tremendous impact on the photo characteristics. Assuming a static environment, an 85mm f2.0 on a full frame camera will have a similar field of view and depth of field as that of a 56mm f1.4 on a crop sensor camera. However, if you were to mount that same 85mm f2.0 lens onto a crop sensor camera, you would actually get the field of view and depth of field of an equivalent full frame 127.5mm f3.0 lens. If the photographer wanted to match the blocking of the 85mm full frame, they would either need to change to a 56mm f1.4 lens or physically need to move the camera back according to the crop factor. With that said, if you were to do the reverse and mount the 56mm f1.4 crop sensor lens onto a full frame camera, you would get the field of view and depth of field of an equivalent full frame 56mm f1.4 lens. The problem with mounting a crop sensor lens onto a full frame camera is that the lens will fail to cover the full frame sensor and you will likely get strong vignetting, leading up to complete blackness outside the lens coverage.*

35MM EQUIVALENT DEPTH OF FIELD The depth of field for a lens does not vary with crop factor. However, If a photographer attempts to capture the same blocking across two different cameras with differing sensor sizes, the depth of field will be the "true" depth of field multiplied by the crop factor of the sensor. The depth of field will only vary based on the true f-stop of the aperture (not the 35mm equivalent aperture) and the distance to the subject. However, the crop factor of a camera will change the field of view for a lens on that camera. For smaller sensors, with a positive crop factor, the 35mm

equivalent focal length will be larger than the true focal length, this means that the field of view will be narrower. Maintaining the same blocking in the shot requires moving your subjects farther back. This in turn will increase the depth of field. So, maintaining consistent blocking will alter the depth of field by the crop factor. 35mm equivalent depth of field and 35mm equivalent focal length are depicted in Figure 1-8.

35MM EQUIVALENT APERTURE A useful but misleading term for the f-stop of the lens multiplied by the crop factor of the photosensitive region. The f-stop of an aperture encodes two things: the depth of field and the exposure. Lenses are frequently referred to by their 35mm equivalent aperture, but this can be slightly misleading. For example, an f1.8 on a crop sensor may be said to have an f2.7 35mm equivalent, due to the 1.5 times crop factor. But this is misleading because the depth of field will increase by the crop factor (if maintaining shot blocking), but the exposure will not vary. So, an f1.8 on a crop sensor will have the depth of field (if maintaining shot blocking) of an equivalent f2.7 lens, but will have the exposure of an equivalent f1.8 lens.

1.10 PHYSICAL LENSES

PRIME LENS A lens designed and marketed as having a fixed focal length.

ZOOM LENS A lens designed and marketed as having a controllable focal length.

ULTRA-WIDE ANGLE LENS A lens with a 35mm equivalent focal length less than 24 mm.

WIDE ANGLE LENS A lens with a 35mm equivalent focal length between 24 mm and 35 mm.

STANDARD LENS A lens with a 35mm equivalent focal length between 35 mm and 60 mm.

TELEPHOTO LENS A lens with a 35mm equivalent focal length between 60 mm and 200 mm.

SUPER TELEPHOTO LENS A lens with a 35mm equivalent focal length greater than 200 mm.

MINIMUM FOCUSING DISTANCE The closest distance from the sensor plane to which a lens can focus, or the distance from the sensor plane to the nearest plane of focus.

MACRO LENS A lens designed and marketed as having a small minimum focusing distance. Macro lenses are described by their magnification ratio at minimum focusing distance. A macro lens with a 1:1 magnification ratio will project an object at perfect scale onto the sensor; e.g., a macro lens with a 1:1 magnification will project a 20-mm-wide object as a 20-mm projection onto the sensor plane.

LENS COVERAGE The diameter of the image circle that is projected onto the sensor plane by the lens. The area outside of the image circle is guaranteed to vignette and is possibly completely black, the area inside of the image circle is the projected image. A lens with a 43mm coverage will cover a full frame photographic sensor with a complete projection. A lens with a 29mm coverage will cover a cropped sensor with a complete projection, but will fail to cover a 35mm photographic sensor such that there will be heavy vignetting leading to total blackness at the periphery. It is generally undesirable to use lenses that do not cover your sensor size.

1.11 BOKEH

BOKEH The aesthetic quality of the out-of-focus regions of an image, pronounced "bo-kay" from the Japanese word for *blur*. Typically used to describe the appearance of point lights that are outside the plane of focus, bokeh technically describes the appearance of all features outside the plane of focus. Bokeh can describe the shape of the circle of confusion, where the "circle" may actually be an *N*-gon or a cat's eye. Bokeh also describes how the circles of confusion for any adjacent features in a scene will combine once projected onto the image sensor. The softness of a lens's bokeh is defined by the distribution of light in the circle of confusion. The three softnesses of bokeh are depicted in Figure 1-9.



(a) *Under-corrected bokeh* **(b)** *Corrected bokeh* **(c)** *Over-corrected bokeh*

Figure 1-9. *A portrait-style image depicted with the three kinds of softness for bokeh. Notice how two-dimensional the over-corrected bokeh background looks compared to the under-corrected bokeh background.*

CORRECTED BOKEH The most common bokeh type found in consumer lenses where the photons for the circle of confusion are uniformly distributed throughout the circle of confusion. Point lights appear as flat disks of light.

OVER-CORRECTED BOKEH The photons for the circle of confusion are predominantly from the edge of the circle. Point lights appear as soap bubbles, colloquially called soap-bubble bokeh. The out-of-focus areas are considered to be quite distracting. Today, over-corrected bokeh is seen as more of an aesthetic choice, rather than being strictly desirable. Lots of old lenses exhibit over-corrected bokeh and presenting over-corrected bokeh in your shot is an effective means of pretending that a scene was shot in the past.

UNDER-CORRECTED BOKEH The photons for the circle of confusion are predominantly from the center of the circle. It is often described as having a "diffuse" quality. The out-of-focus areas lack sharp features or contrast and reinforce a separation between in-focus and out-of-focus regions of an image.

N-BLADE APERTURE (WHERE N IS A NUMBER) The variable aperture in most lenses changes its diameter by rotating a set of N concave blades closer to the lens center (shrinking the aperture) or closer to the lens wall (expanding the aperture). This can cause the out-of-focus regions of a camera to appear as N-gons instead of perfect circles.

CAT'S-EYE BOKEH The out-of-focus areas of an image take on the shape of a cat's eye instead of a circle. Cat's eye bokeh predominantly come from two different effects. For lenses with aspherical elements, the bokeh at the periphery of the image can be laterally compressed, producing out-of-focus areas that appear as ellipses. The other predominant form of cat's eye occurs when the circle of confusion for a feature falls partially outside of the entrance pupil or the exit pupil. Instead of a perfect circle, the bokeh appears as the intersection of two circles, due to some of the light falling outside of a pupil, leading to a shape that is the overlapping projection of two circles. The shape associated with the intersection of two circles is actually called a lens.

There is a very simple means of controlling the quality of bokeh for a virtual scene. When choosing an initial random ray direction for a thin lens model, rather than sampling randomly from the unit disk, sample randomly from a bokeh texture. This is just a two-dimensional grayscale picture that represents the shape and probability of the aperture. A perfectly corrected bokeh disk would just be a gray circle within a black background. An over-corrected bokeh would be a gray circle that tapers toward max white at

the edges, whereas an under-corrected bokeh would be a gray circle that tapers toward black at the edges. This can also be used to simulate shapes, such as the common *N*-gon and cat's eye, or can be used for more exotic shapes, such as stars and spirals.

1.12 VARIOUS LENS IMPERFECTIONS

FOCUS BREATHING An undesired effect where the focal length of a lens will change as the lens changes focus. For many lenses focusing closer to the camera increases the focal length (zooming in), and focusing farther from the camera decreases focal length (zooming out).

PARFOCAL A parfocal lens will maintain the same plane of focus as the user changes the focal length. This is desirable in videographic work where the videographer can zoom in on their subject and maintain sharp focus.

VARIFOCAL A varifocal lens will shift its plane of focus as the user changes the focal length. This is undesirable in videographic work.

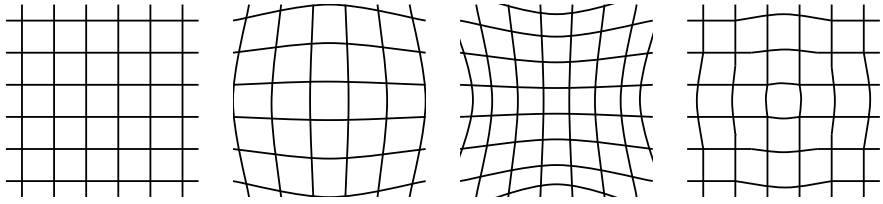
VIGNETTING A drop in exposure from the center of the image to the edges of the image. This is caused by a drop in photons captured at the frame of an image due to blockage by mechanical elements or incomplete coverage by any of the individual optical elements.

Vignetting can be cheaply modeled as a post-processing effect by selecting at what angle (from the longitudinal axis) or what radius (from the center of the image) vignetting starts, at what angle or radius vignetting ends, the darkness at the end of vignetting, and the interpolation function between the two extremes. More than two anchor points can be used for a more complex vignetting pattern. This same technique can also be used to counteract vignetting on physical lens systems to produce even exposure across the image, but note that you will have an increased noise floor where the image was raised up.

RECTILINEAR LENS A lens that projects straight lines in the environment as straight lines on the sensor plane. (See Figure 1-10a.)

CURVILINEAR LENS A lens that projects straight lines in the environment as curved lines on the sensor plane.

BARREL DISTORTION The distortion where a curvilinear lens projects straight lines in the environment as curved lines that bow outward away from the image center. (See Figure 1-10b.)



(a) *Rectilinear* **(b)** *Barrel* **(c)** *Pin cushion* **(d)** *Mustache*

Figure 1-10. *A Cartesian grid drawn with no major distortion, with barrel distortion, with pin cushion distortion, and with both (mustache distortion).*

PIN CUSHION DISTORTION The distortion where a curvilinear lens projects straight lines in the environment as curved lines that bow inward toward the image center. (See Figure 1-10c.)

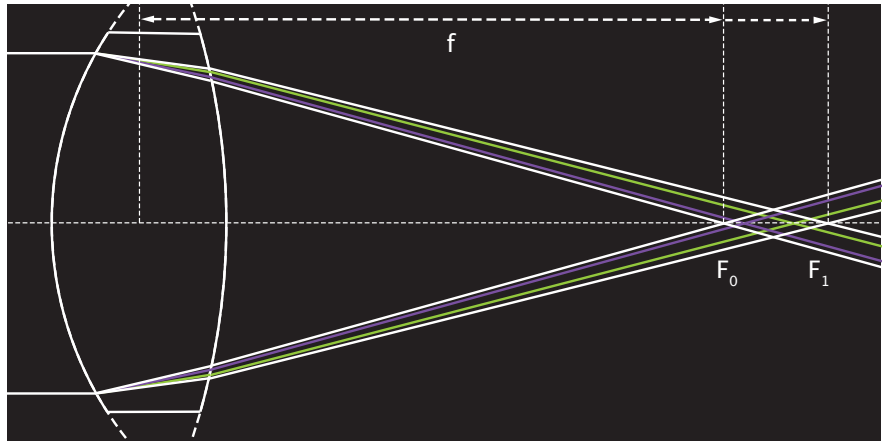
MUSTACHE DISTORTION The distortion where a curvilinear lens projects straight lines in the environment as curved lines that exhibit both barrel and pin cushion distortion, creating curving lines that form in waves reminiscent of a mustache. (See Figure 1-10d.)

FISHEYE LENS A lens that attempts to capture the entire hemisphere in front of the lens. Fisheye lenses will have significant barrel distortion as they map a hemisphere onto a plane.

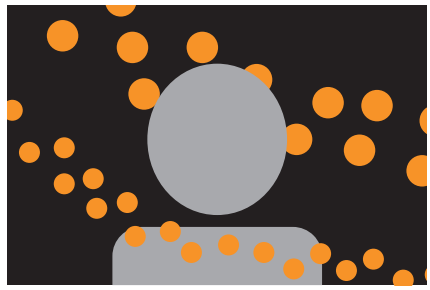
FLARING A smear of light in the captured image due to a bright light source incident to the lens. Instead of only being captured at the projected location of the light, the bright spot will bounce either off of glass elements inside the lens or off of the lens wall and produce a visible bright smear somewhere in the captured image. The light source does not need to be within the lens projection, its photons only need be incident to the lens. This can be a source of lost contrast in an image.

Many types of lenses exhibit unique and interesting flaring. This can be cheaply simulated as a post-process on a final image by identifying places of high exposure and producing your desired flare shapes. The big problem with this approach is that it can fail to capture the angle of incidence for those spots of bright exposure, as the flaring characteristic of lenses is best characterized by their response to a collimated beam of light from various angles.

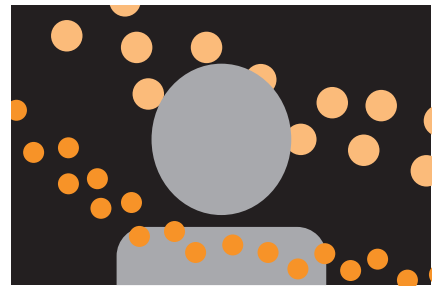
SPHERICAL ABERRATION Imperfect lensing caused by spherical optical elements. The incoming parallel lines do not correctly converge to a single point, and it appears as softness in a captured image.



(a) *Longitudinal lensing error, where incident photons with differing wavelengths will be lensed by differing foci along the longitudinal axis*



(b) *No longitudinal chromatic aberration*

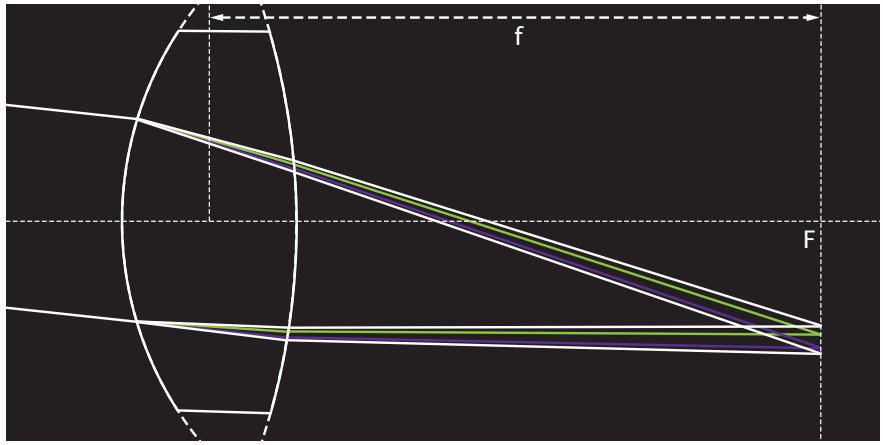


(c) *With longitudinal chromatic aberration*

Figure 1-11. *Longitudinal chromatic aberration is caused by differing indices of refraction across the electromagnetic spectrum. A thin lens will exhibit severe loca as the single optical element is guaranteed to act as a prism. In this example, differing focal lengths causes features closer than the plane of the focus to shift purple—shifting the orange point lights toward pink—and features farther than the plane of focus to shift green—shifting the orange point lights toward yellow.*

LONGITUDINAL CHROMATIC ABERRATION Imperfect lensing that causes different wavelengths of light to have differing focal lengths, colloquially referred to as loca. The foci of differing wavelengths vary along the longitudinal axis of the lens. This can cause a chroma shift for objects that do not fall directly on the plane of focus. An example of the lensing and its effect on images can be seen in Figure 1-11.

TRANSVERSE CHROMATIC ABERRATION (COLOR FRINGING) Imperfect lensing that causes gradations of chroma or luma (known as fringing) along borders of high contrast, also called lateral chromatic aberration. The



(a) *Transverse lensing error, where incident photons at an oblique angle with differing wavelengths will be projected onto differing radii along a transverse line*



(b) *No transverse chromatic aberration*



(c) *With transverse chromatic aberration*

Figure 1-12. *Transverse chromatic aberration is caused by differing the radius of photon convergence across the electromagnetic spectrum. The photons all converge on a single plane orthogonal to the longitudinal axis, but converge at differing points along a transverse line on that plane. This can cause visible color fringing at borders of high contrast. In this example, transverse chromatic aberration causes noticeable purple and green fringing at borders of high contrast in the corners of the image. It is common for there to be no color fringing in the highlights due to the highlights being clipped. This figure has its background set to just under clipping so that green fringing is still present.*

wavelengths for all light coming from an oblique angle to the front element will all converge on the plane of the focal point, but each wavelength will converge at a differing radius from the focal point. The foci for oblique lines will vary laterally (along the plane) by wavelength. Transverse chromatic aberration is depicted in Figure 1-12.

FIELD CURVATURE The shape and orientation of the "plane" of focus. The plane of focus is never truly perpendicular to the longitudinal axis of the

lens in a physical lens, and it is also never a true plane but actually a curved surface.

This can be simulated by changing how the simulated optical elements are modeled. The plane of focus can be rotated by rotating the thin lens model's unit disk off of the longitudinal axis of the lens. The plane of focus can be curved by mapping the thin lens model's unit disk onto a curved surface.

DIFFRACTION An undesirable effect leading to reduced sharpness in physically captured images. Diffraction in photography almost always refers to the reduction in sharpness at high *f*-numbers (small apertures), where the effects of wave diffraction can create interference that is larger than the pixel pitch.

1.13 OPTICAL ELEMENTS

OPTICAL ELEMENT A single part of the lensing apparatus, usually made of glass, with a defined shape and size.

SPHERICAL ELEMENT An optical element whose sides are either flat or model a spherical curve.

ANAMORPHIC ELEMENT An optical element with one or both sides that are cylindrical.

ASPHERICAL ELEMENT An optical element that is rotationally symmetric about the longitudinal axis, but which has one or both sides that are not flat, spherical, or cylindrical.

SPHERICAL LENS A lens that only contains spherical elements.

ASPHERICAL LENS A lens that only contains spherical and aspherical elements. Wide angle and ultra-wide angle lens frequently use aspherical elements to optimize for the wide field of view.

COATING The coating that is applied to optical elements to achieve certain optical qualities. Coatings are applied for such technical reasons as altering the incident index of refraction or altering the wavelengths that are absorbed. Coatings can be applied for aesthetic reasons such as controlling contrast, sharpness, or flaring.

1.14 ANAMORPHIC

ANAMORPHIC LENS A lens that contains one or more anamorphic elements. An anamorphic lens is defined by its squeeze ratio, typically between 1.3:1 and 2:1. An anamorphic lens will squeeze the horizontal dimension during image capture, and the image will need to be desqueezed in post-production. A 2:1 anamorphic lens with a 50mm focal length will maintain the vertical field of view of a 50mm lens, but will have the horizontal field of view of a 25mm lens. The cylindrical elements occasionally produce horizontal flaring in the image. Anamorphic lenses will have oval bokeh. They are synonymous with Hollywood filmmaking due to their use in many popular films.

1.15 CAMERA MOVEMENT

PAN A rotation of the camera about its vertical axis.

TILT A rotation of the camera about its horizontal axis.

ROLL A rotation of the camera about its forward axis.

TRUCK A translation of the camera along its horizontal axis.

DOLLY A translation of the camera along its forward axis.

BOOM A translation of the camera along its vertical axis.

TRACK A translation of the camera along an arbitrary plane, usually following a subject.

REFERENCES

[1] Bayer, B. E. Color imaging array. Patent (US3971065A). 1976.

[2] Buettner, C. Image processing method and filter array. Patent (US20190306472A1). 2019.

[3] CIE. *Commission Internationale de l'Eclairage Proceedings*. Cambridge University Press, 1931.

[4] Fujifilm. Fujifilm X-Trans sensor technology. https://www.fujifilm.eu/uk/products/digital-cameras/model/x-pro1/features-4483/aps-c-16m-x-trans-cmos, 2017.

[5] Gilder, G. *The Silicon Eye: How a Silicon Valley Company Aims to Make All Current Computers, Cameras and Cell Phones Obsolete*. W. W. Norton & Company, 2005.

[6] ISO. 12232:2019: Photography—Digital still cameras—Determination of exposure index, ISO speed ratings, standard output sensitivity, and recommended exposure index. https://www.iso.org/standard/73758.html, 2019.

[7] ISO. 15739:2017: Photography—Electronic still picture imaging—Noise measurements. https://www.iso.org/standard/72361.html, 2017.

[8] Langlands, A. Physlight. https://github.com/wetadigital/physlight, 2020.

[9] Schanda, J. *Colorimetry: Understanding the CIE System*. John Wiley & Sons, Inc., 2007.

[10] Sony. Sony releases stacked CMOS image sensor for smartphones with industry's highest 48 effective megapixels. *Sony Global*, https://www.sony.com/en/SonyInfo/News/Press/201807/18-060E/, July 23, 2018.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 2

RAY AXIS-ALIGNED BOUNDING BOX INTERSECTION

Peter Shirley, Ingo Wald, and Adam Marrs
NVIDIA

ABSTRACT

The slabs test is the dominant approach to determine if a ray hits an axis-aligned bounding box (axis-aligned 3D rectangle that encloses a region). There is a particularly clean way to code this algorithm that is widely used but not singled out very well in the literature, and this chapter describes it.

2.1 THE METHOD

Though most intersection tests find where a ray hits an object, along with information such as surface normals and material properties, for ray tracing acceleration structures, we often just need to know whether or not a ray hits a bounding object. A common one is an *axis-aligned bounding box* (AABB), the 3D analog of a rectangle. There has been a plethora of different tests devised to do this, but the dominant method is the *slabs test* of Kay and Kajiya [2]. Their algorithm takes a ray defined by an origin $\mathbf{o}$ and direction $\mathbf{v}$, $\mathbf{p}(t) = \mathbf{o} + t * \mathbf{v}$, and decides whether or not the ray hits an axis-aligned bounding box with corners $\mathbf{p}_0$ and $\mathbf{p}_1$ by computing the ray's intersections with three "slabs" and then seeing if it is ever within all three slabs at once. A *slab* is the region of space enclosed by two parallel planes. Kay and Kajiya's insight is that the intersection of three slabs is a volume, and if a ray hits a volume, it must be simultaneously inside all three slabs at once.

In the case of an AABB, we can define the bounding box as the intersection of three slabs, each parallel to one of the cardinal axes. The x-slab is the region of space sandwiched between the two planes $x = x_0$ and $x = x_1$. This is an infinite thing with finite thickness like an infinite piece of drywall or steel plate. If the ray is not parallel to the slab, then it will hit the slab boundaries at t-values t_{x0} and t_{x1}. The ray is *inside* the slab for the interval $t \in [t_{x0}, t_{x1}]$. Note that there are analogous intervals for y and z and each of these intervals could be increasing or decreasing. Using the slabs test, we see that if there is

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_2

any overlap of all three intervals, the *t*-values will indicate that the ray is inside the bounding box, so an "is there an overlap?" test is the same as a "does the ray hit?" test. A nice aspect of framing the test in terms of interval overlap is that most ray tracers have a `[tstart, tfinish]` interval of valid hit distances, which can be rolled into the slabs test.

Almost all implementations of the slabs test follow the same patten and differ only in details. As discussed by Majercik et al. [4], various practitioners [1, 3] have found a natural and compact way to express the slabs algorithm in terms of vector syntax. We present that here in a version that includes the *t*-interval as a fourth interval overlap test.

```
// boxLower and boxUpper are the minimum and maximum box
// corners; the interval [rayTmin,rayTmax] is the interval of
// the t-values on the ray that count as a hit; invRayDir
// is (1/vx,1/vy,1/vz), where v is the ray direction.
bool slabsBoxTest(/* box to test ray against */
                  vec3 p0, vec3 p1,
                  /* the ray to test against the box */
                  vec3 rayOrigin, vec3 invRayDir,
                  real rayTmin, real rayTmax)
{
   // Absolute distances to lower and upper box coordinates
   vec3 tLower = (p0 - rayOrigin)*invRaydir);
   vec3 tUpper = (p1 - rayOrigin)*invRaydir);
   // The four t-intervals (for x-/y-/z-slabs, and ray p(t))
   vec4 tMins  = (min(tLower,tUpper), rayTmin);
   vec4 tMaxes = (max(tLower,tUpper), rayTmax);
   // Easy to remember: ``max of mins, and min of maxes''
   real tBoxMin = max_component(tMins);
   real tBoxMax = min_component(tMaxes);
   return tBoxMin <= tBoxMax;
}
```

Note that this code will produce the correct answer for ray directions that contain zero components provided the underlying hardware is IEEE floating-point compliant (see Williams [5] for an explanation). A caution on the code above is that it will return `false` for the case of a ray origin on the box and a ray direction in a box face (where a NaN will occur), and if this is not desired behavior, the code will need to be modified.

REFERENCES

[1] Áfra, A. T., Wald, I., Benthin, C., and Woop, S. Embree ray tracing kernels: Overview and new features. In *ACM SIGGRAPH 2016 Talks*, 52:1–52:2, 2016.

[2] Kay, T. L. and Kajiya, J. T. Ray tracing complex scenes. In *Proceedings of SIGGRAPH*, pages 269–278, 1986.

[3] Laine, S., Karras, T., and Aila, T. Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 137–143, 2013.

[4] Majercik, A., Crassin, C., Shirley, P., and McGuire, M. A ray-box intersection algorithm and efficient dynamic voxel rendering. *Journal of Computer Graphics Techniques*, 7(3):66–81, 2018.

[5] Williams, A., Barrus, S., Morley, R. K., and Shirley, P. An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools*, 10(1):49–54, 2005.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 3

ESSENTIAL RAY GENERATION SHADERS

Morgan McGuire and Zander Majercik
NVIDIA

ABSTRACT

The most fundamental step in any ray tracing program is the ray generation (or "raygen") program that produces the rays themselves. This chapter describes seven essential projections that should be in every toolkit and gives parameterized raygen shader code for each.

3.1 INTRODUCTION

The ray generation projections we present are useful as primary (a.k.a. view or eye) rays for a camera as well as for light probe computation and photon emission. Many of these were not previously available in a convenient, drop-in raygen form but were instead published as image space transformations or rasterization pipeline vertex programs. We discuss the following essential projections:

- *Pinhole (perspective):* Standard/rectilinear/planar perspective projection.
- *Thin lens:* Gaussian lens model for generating in-camera depth of field.
- *Generalized Panini:* Better than pinhole for ultrawide displays.
- *Fisheye:* Traditional wide field of view for pinhole projection.
- *Lenslet:* Light field slab parameterization; grid of fisheyes.
- *Octahedral:* Good for reflection probes.
- *Cube map:* Both texel-to-ray and ray-to-texel transformations.
- *Orthographic (parallel):* Good for scientific and engineering visualization.
- *Fibonacci uniform sphere:* Good for ambient occlusion and radiosity.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_3

For each projection, we provide the shader code, an image showing the projection applied to a 3D scene, and a discussion of when the projection is useful. We also include subpixel offset code for supersampling using a Halton sequence. We begin with a discussion of the shared camera-to-world transformation and diagrams showing the ray generation model and parameters. We release all code from this chapter into the public domain so that it can be used without restriction.

3.2 CAMERA RAYS

3.2.1 CAMERA SPACE

For simplicity, we refer to the origin space of the rays as the mathematical *camera space* and reference a *viewer* where there is an image plane. There may not be a camera in the real-world sense—for example, if rendering light probes.

The shaders in this chapter generate rays in camera space, using the convention that $\hat{x}$ = right, $\hat{y}$ = up, and $\hat{z}$ = toward the viewer. Because the ray tracing acceleration structure is typically built in world space, these rays must be transformed from camera space to world space for tracing. We present this common transformation code here and omit it from the individual ray generation shaders in the remainder of the chapter.

Let a ray be defined by:

```
#ifndef GLSL // For HLSL, OptiX, and CUDA
#   define vec3 float3
#   define mat4x3 float4x3
#   define mat3x3 float3x3
#endif

struct Ray {
    vec3  pos;  // Origin
    float min;  // Distance at which intersection testing starts
    vec3  dir;  // Direction (normalized)
    float max;  // Distance at which intersection testing ends
};
```

We interleave the distances with the `vec3` types because when working with large numbers of rays or rays stored in buffers, this gives better 4-vector alignment in memory.

Camera orientations can be parameterized in many ways, but we always assume that the camera's axes are orthonormal; the camera itself is not somehow skewed or stretched.

When the camera's orientation (i.e., camera-to-world transformation) is described by a `mat3x3` rotation matrix `cameraRotation` whose columns are the axes of the camera space and a `vec3` center of projection `cameraTranslation`, the camera-to-world transformation is:

```
1 worldRay.dir = matmul(cameraRotation, cameraRay.dir);
2 worldRay.pos = cameraTranslation + cameraRay.pos;
```

If the camera's orientation is instead packed into a single `mat4x3` `cameraToWorld` matrix, then this transformation becomes:

```
1 worldRay.dir = matmul(mat3x3(cameraToWorld), cameraRay.dir);
2 worldRay.pos = cameraToWorld[3] + cameraRay.pos;
```

Finally, when the camera is described by a `mat4x3` world-to-camera matrix, the transformation is:

```
1 worldRay.dir = matmul(cameraRay.dir, mat3x3(worldToCamera));
2 worldRay.pos = worldToCamera[3] + cameraRay.pos;
```

Note that we adhere to the OpenGL column-major naming convention: `mat4x3` means "4 columns, 3 rows."

3.2.2 NEAR AND FAR PLANES

Rasterization generally requires a near z clipping plane. The alternative is very careful handling of the singularity at $z = 0$, for which many engines are not prepared. Rasterization also typically employs a far z clipping plane, although the far plane is not mathematically required. In fact, better precision is available for nearby surfaces under depth buffering when taking the limit of the projection matrix as the far plane moves to an infinite distance [2]. However, a finite far clipping plane is very useful for limiting the number of visible primitives. Because rasterization in particular requires linear time in the number of primitives, that is an important performance optimization.

Ray tracing does not mathematically require a near or far clipping plane and is easier to implement without them. However, there are several reasons to introduce them. Ray tracing performance generally decreases as the length of the ray increases, until the ray leaves the bounding box of the scene. It is often useful to integrate ray tracing and rasterization into the same pipeline. In this case, the ray tracer's results should match the near and far clipping of the rasterization pipeline. Finally, when an object comes very close to the camera, it can become arbitrarily large and confuse the viewer. A real camera cannot get arbitrarily close to an object because the camera itself takes up space, so this should be resolved by the physics or UI parts of the graphics

system. When those are too simple or fail, it is sometimes helpful to clip a nearby object instead of filling the screen. It is also sometimes useful to clip nearby objects, to peek inside of them in either engineering or debugging contexts. Of course, it is also sometimes useful to *not* clip nearby objects, for example to prevent cheating in games by peering through a wall.

When clipping is desirable, the best way to implement it is by modifying the `ray.min` and `ray.max` instead of moving the ray's origin. This preserves precision and is already accounted for by the ray traversal and intersection hardware.

For spherical and octahedral projections and for some cube maps, the clipping "plane" is usually a sphere, so set the min and max as:

```
ray.min = abs(zNear);
ray.max = abs(zFar);
```

We insert the absolute value because graphics systems have differing sign conventions for clipping plane parameters. The numbers used here must be nonnegative.

For the other projections and some cases of cube maps, clipping is to a plane. But because rays are generally not along the camera-space $\hat{z}$ axis, the clipping distance differs for each ray and is not the same as the clipping plane depth. The ray approaches the clipping plane at a rate that is the inverse of the dot product of `rayDir` and the camera-space $\hat{z}$ axis, which is simply the `z` component of the direction:

```
ray.min = abs(zNear) / ray.dir.z;
ray.max = abs(zFar) / ray.dir.z;
```

When clipping is not desirable, simply set the near plane to a distance of zero and the far plane to infinity:

```
#ifdef GLSL
    // Positive infinity IEEE 754 float32
    const INFINITY = uintBitsToFloat(0x7F800000);
#elif HLSL
    const INFINITY = asfloat(0x7F800000);
#else // CUDA/OptiX
#   include <cmath>
#endif

ray.min = 0.f;
ray.max = INFINITY;
```

We use the bitwise representation of infinity to work around the limitation that GLSL and HLSL have no standardized infinity or maximum float

Listing 3-1. *DirectX MSAA subpixel offsets, with pixel center* (0.5, 0.5)*. (From [10].)*

```
const vec3 s1[]  = {{0.5f, 0.5f}};
const vec3 s2[]  = {{0.25f, 0.25f}, {0.75f, 0.75f}};
const vec3 s4[]  = {{0.375f, 0.125f}, {0.875f, 0.375f},
                    {0.625f, 0.875f}, {0.125f, 0.625f}};
const vec3 s8[]  = {{0.5625f, 0.6875f}, {0.4375f, 0.3125f},
                    {0.8125f, 0.4375f}, {0.3125f, 0.8125f},
                    {0.1875f, 0.1875f}, {0.0625f, 0.5625f},
                    {0.6875f, 0.0625}, {0.9375f, 0.9375f}};
const vec3 s16[] = {{0.5625f, 0.4375f}, {0.4375f, 0.6875f},
                    {0.3125f, 0.375f}, {0.75f, 0.5625f},
                    {0.1875f, 0.625f}, {0.625f, 0.1875f},
                    {0.1875f, 0.3125f}, {0.6875f, 0.8125f},
                    {0.375f, 0.125f}, {0.5f, 0.9375f},
                    {0.25f, 0.875f}, {0.125f, 0.25f},
                    {0.0f, 0.5f}, {0.9375f, 0.75f},
                    {0.875f, 0.0625f}, {0.0625f, 0.0f}};
```

literal/constant, although vendors support various extensions such as `1.#INF`. For OpenGL 4.1 and later, division by zero will also generate the correctly signed infinity.

3.2.3 SUPERSAMPLING

Averaging the contributions of multiple rays per pixel or texel reduces aliasing artifacts. This is the strategy that multisample antialiasing (MSAA) [4] and temporal antialising (TAA) [12] use under rasterization. For ray tracing, the rays must be offset in image space, which changes their directions but not origins (except for orthographic cameras, which are the opposite).

To apply a subpixel offset to each ray, we just change the fractional pixel coordinate before computing the ray itself. The offsets may be chosen by hand or according to a mathematical pattern. The standard DirectX MSAA subsampling patterns for 1, 2, 4, 8, and 16 samples per pixel are given in Listing 3-1. A common pattern for an arbitrary number of low-discrepancy subpixel offsets is the 2,3 Halton sequence in Listing 3-2. Because it runs in $O(N \log N)$ time in the number of samples, this should be precomputed on the CPU for the desired number of samples and then passed to the ray generation shader.

3.2.4 VIEW CAMERAS

Consumer real-world cameras today place the image plane orthogonal to the view axis. Professional photographers may also use *view cameras*, which have

Listing 3-2. *C code for generating* `N` *Halton samples.*

```
void generateHaltonSequence(int N, int b, float sequence[]) {
    int n = 0, d = 1;
    for (int i = 0; i < N; ++i){
        int x = d - n;
        if (x == 1) {
            n = 1;
            d *= b;
        } else {
            y = d / b;
            while (x <= y) {
                y /= b;
            }
            n = (b + 1) * y - x;
        }
        sequence[i] = (float)n / (float)d;
    }
}

void generateSubpixelOffsets(int N, Vector2 offset[]) {

    float xOffset[N];
    float yOffset[N];

    generateHaltonSequence(N, 2, XOffset);
    generateHaltonSequence(N, 3, YOffset);

    for (int i = 0; i < N; ++i) {
        offset[i].x = XOffset[i] - 0.5f;
        offset[i].y = YOffset[i] - 0.5f;
    }
}
```

an additional degree of freedom for rotating and translating the image plane relative to the view axis and center of projection [8]. This physical design was once more common and is why old-fashioned large format cameras have an accordion-pleated bellows objective. A view camera allows distorted perspective projection and tilt/shift focusing. Today, these effects are often simulated in digital post-processing. A ray tracer can generate these effects in camera by modifying the ray generation shader. Because it is uncommon, we do not detail the transformations here, but the general idea is simple: transform the 3D camera-space virtual pixel sampling position before computing the ray to it from the center of projection.

3.2.5 PARAMETERS

To reduce the size of each code listing, Listing 3-3 gives the set of all parameters for all projections as global variables. In most cases, some

Listing 3-3. *The global parameters used by subsequent listings in this chapter.*

```
// Edge-to-edge field of view, in radians
float cameraFOVAngle;

// 0 = cameraFOV is the horizontal FOV
// 1 = vertical
// 2 = diagonal (only used for fisheye lens)
int cameraFOVDirection;

// Integers stored as floats to avoid conversion
vec2 imageSize;

// Center of projection from cylinder to plane,
// can be any positive number
float paniniDistance;

// 0-1 value to force straightening of horizontal lines
// (0 = no straightening, 1 = full straightening)
float paniniVerticalCompression;

// Scalar field of view in m, used for orthographic projection
float cameraFovDistance;

// Lens focal length in meters,
// would be measured in millimeters for a physical camera
float lensFocalLength;

// Ratio of focal length to aperture diameter
float fStop;

// Distance from the image plane to the lens
// The camera is modeled at the center of the lens.
float imagePlaneDistance;
```

expressions of parameters such as the tangent of the field of view can be computed on the host outside the shader to avoid expensive operations per ray. We give the straightforward shaders here so that the derivations are clear and trust the implementor to recognize these optimization opportunities.

3.3 PINHOLE PERSPECTIVE

Pinhole perspective projection simulates a camera with a pinhole aperture (i.e., approaching a zero radius). It is the common rectilinear perspective projection used for rasterization graphics and by artists for most perspective line drawings with a small field of view. (See Figure 3-1.)

The "multi-point" perspective artists refer to, such as 1-point perspective or 2-point perspective, is about techniques for executing pinhole projection by hand and has no mathematical significance for the projection itself. The

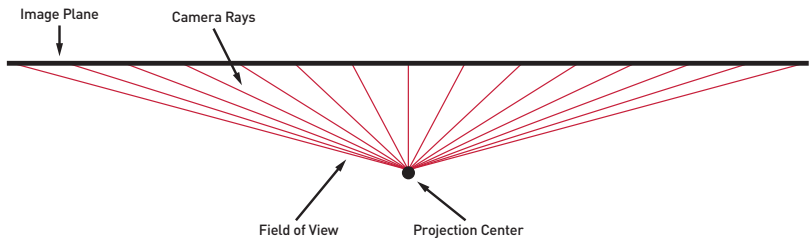


Figure 3-1. *Top-down labeled view of a pinhole perspective projection (150° field of view). Note the significant angular distortion at the extreme edges of the field of view.*

points in the context are the places where sets of 3D parallel lines converge to a point on the image plane; there are always an infinite number of such points. The only reason that artists identify a small number of them is that those are the dominant lines of their composition—for example, the lines along the edges of buildings or roads.

A lens is not required for image formation. The aperture creates an image. When it is a pinhole, the image resolves well. However, the amount of light that passes through is also small, which makes real pinhole camera images very faint (and noisy). To use such a camera, the photographer must compensate with a very long exposure time. That in turn requires a static scene to minimize motion blur. In computer graphics, we can scale the intensity of the image independently from exposure time and aperture size, so a computer graphics pinhole camera can produce an image that is perfectly in focus everywhere, bright, and free of photon noise. Such image transformations are more nuanced with a physical camera (see Chapter 1).

Listing 3-4 gives the pinhole projection, which produces images such as Figure 3-2. It is useful for exactly matching a rasterization pipeline. Note that `-pixel.y` corrects for the screen space being upside down (bottom left to top right as opposed to top left to bottom right).

Pinhole projection preserves straight lines and scales objects at equal distances along the view axis by equal amounts. When a physical display is flat and exactly matches the virtual display parameters including the position of the viewer, pinhole projection gives the correct projection for an eye staring straight ahead to perceive the display as a window into the virtual world, except that there is no defocusing due to depth or stereo vision.

Listing 3-4. *Pinhole perspective projection.*

```
// Pixel is the integer position >= 0 and < imageSize.
Ray generatePinholeRay(vec2 pixel) {
    float tanHalfAngle = tan(cameraFovAngle / 2.f);
    float aspectScale = (cameraFovDirection == 0) ? imageSize.x : imageSize.
        y;
    vec3 direction = normalize(vec3(vec2(pixel.x, -pixel.y) * tanHalfAngle /
         aspectScale, -1));
    return Ray(vec3(0.f), direction, 0.f, INFINITY);
}
```



Figure 3-2. *Example of pinhole perspective projection.*

When the viewer looks away from the center of the display or moves their head from the center of projection, the projection is incorrect for what they would see if the display were a window. Because the human visual system is adaptive, most people usually do not notice that the projection is incorrect in this case and cannot only understand the scene but still feel comfortable. All 2D artwork and computer graphics has always relied on this. We can change the parameters, such as presenting a wider field of view than the display really subtends, and the image is still acceptable. However, if the field of view is very wide or is curved, the necessary distortion for creating a window-like view for a centered eye can make the image unacceptable to a non-centered one.

For a field of view greater than about 110° on ultrawide 32:9 displays, curved displays, IMAX theatres, projection domes, and so forth, other projections may be better for minimizing motion sickness and shape distortion in the periphery. For large fields of view, the objectionable distortions are that objects are very small in the center of the screen, objects at the edge of the screen appear heavily skewed, and camera rotation causes rapid shape and size distortions between those regions. The pinhole projection is also

inherently limited to a field of view of less than 180°, because at that extent an infinitely large screen is required.

3.4 THIN LENS

A large aperture on a real camera increases the amount of light available and thus shortens the necessary exposure, reducing motion blur. It also defocuses the entire image because the shape of the aperture is convolved with the image, turning all points in the picture into overlapping, blended pictures of the aperture itself. The role of the lens in a camera is to make all rays to a certain depth converge on the image plane, so that one plane can be represented as sharply in focus in the image. Graphics professionals refer to the *circle of confusion* as the 2D region on the image plane that they consider acceptably in focus; for computer graphics, this is approximately the region of a pixel.[1] All other depths have varying blur, which increases based on the distance from that focal plane. The *focus field* is the 3D region that corresponds to the depths producing blur less than the circle of confusion in 2D.

This defocusing is a valuable aesthetic tool for composition because it allows de-emphasizing of the background or extreme foreground framing elements. The *depth of field* is the z-extent of the focus field. A large depth of field is good for landscapes, where both near and far elements should be approximately in focus. A shallow depth of field is good for portraits and conversations, where the audience's attention should be directed to a specific character.

A single, thin lens can produce mathematically ideal focus for a single frequency of light and a small image, with the relationship between the radius (measured with an *f-stop*), curvature (described by a *focal length*, usually measured in millimeters), and distance from the image plane determining the depth of field and in-focus plane. In a real camera, a single lens would create chromatic aberration, barrel distortion from miscalibration or a large field of view, vignetting from the edges of the camera body, and other distortions from imperfect manufacturing. These are effects that graphics systems often simulate in order to make the images match what we are accustomed to seeing in photography or film. Ironically, they are also effects that camera manufacturers seek to minimize. The *objective* of a modern camera that is casually called a "lens" is actually a barrel that contains multiple lenses with various coatings that seek to simulate a single ideal lens.

[1] Photographers have a related but slightly different definition—see Chapter 1.

Listing 3-5. *Lens camera projection.*

```
Ray generateThinLensRay(vec2 pixel, vec2 lensOffset) {

    Ray pinholeRay = pinholeRay(pixel);

    float theta = lensOffset.x * 2.f * pi;
    float radius = lensOffset.y;

    float u = cos(theta) * sqrt(radius);
    float v = sin(theta) * sqrt(radius);

    float focusPlane = (imagePlaneDistance * lensFocalLength) / (
        imagePlaneDistance - lensFocalLength);

    vec3 focusPoint = pinholeRay.direction * (focusPlane / dot(pinholeRay.
        direction, vec3(0.f, 0.f, -1.f)));

    float circleOfConfusionRadius = focalLength / (2.f * fStop);

    vec3 origin = vec3(1.f, 0.f, 0.f) * (u * circleOfConfusionRadius) + vec3
        (0.f, 1.f, 0.f) * (v * circleOfConfusionRadius);

    vec3 direction = normalize(focusPoint - origin);

    return Ray(origin, direction, 0.f, INFINITY);
}
```



Figure 3-3. *Thin lens projection with 128 rays per pixel.*

In graphics, the common practice is to directly model the single ideal lens and then post-process the image to introduce desirable imperfections [1]. The standard model of an ideal lens is called the *Gaussian thin lens model*. It can be parameterized using photographer settings or with scene-based metrics describing the desired effect directly. Listing 3-5 gives the ray generation shader and conversion between the two sets of parameters. It depends on a random position per ray within the lens. Using a single ray per pixel will produce a defocused but noisy image. Averaging multiple rays per pixel

produces a smoother result. We rendered Figure 3-3 using 128 rays per pixel, which was obviously slower than the corresponding image with a single ray per pixel. The cost of multiple rays can be amortized in various ways, including reprojection from previous frames or decoupling rays from pixels and reusing their hit results.

3.5 GENERALIZED PANINI

Above a 70° horizontal field of view, distortions of shapes in a pinhole projection become noticeable. For a very wide (e.g., 160°) field of view, objects at the center of the screen are relatively small, but objects at the sides are very large—so movement between the two regions is disorienting for viewers. For games and flythroughs on ultrawide aspect displays, this is particularly objectionable as the object of interest is usually compressed in the center and the acceleration of objects in the periphery is distracting.

Eighteenth-century artists such as Panini (also spelled Pannini) faced the same problem for wide field of view compositions and developed alternative projections. The Generalized Panini projection [11] can preserve vertical straight lines while approximately preserving shapes and horizontal straight lines throughout the field of view (see Figures 3-4 and 3-5). It produces the most distortion for straight lines near the edges of the vertical field of view, where they become large arcs. Figure 3-6 (center) shows that Panini produces the appearance of a less distorted image than pinhole projection for a wide field of view.

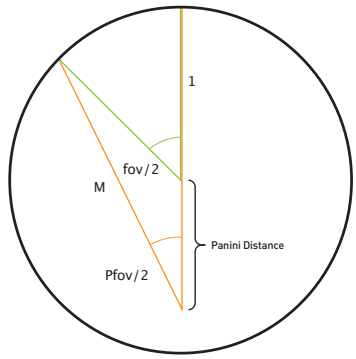


Figure 3-4. *Diagram of the Panini projection. The green lines mark the field of view from the perspective of the camera. The orange lines mark the stretched field of view from the virtual Panini camera that projects the cylinder back to the plane. The length M is used to compute the coordinates of the point on the cylinder to project (see Listing 3-6).*

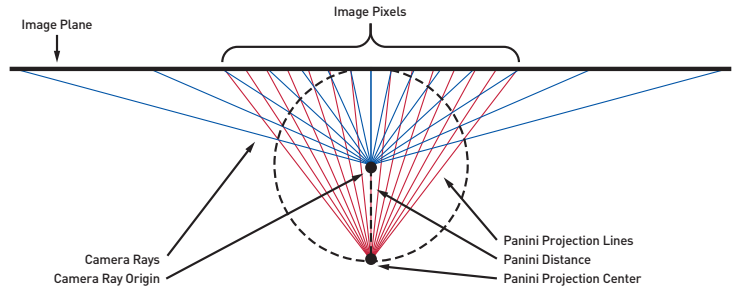


Figure 3-5. *Top-down view of rays generated according to the Panini projection. The second projection is a pinhole projection with a center offset from the camera position. The red lines show the projection of that pinhole onto the image plane. The blue lines show the camera rays that are actually traced into the scene before they are projected onto the image pixels.*

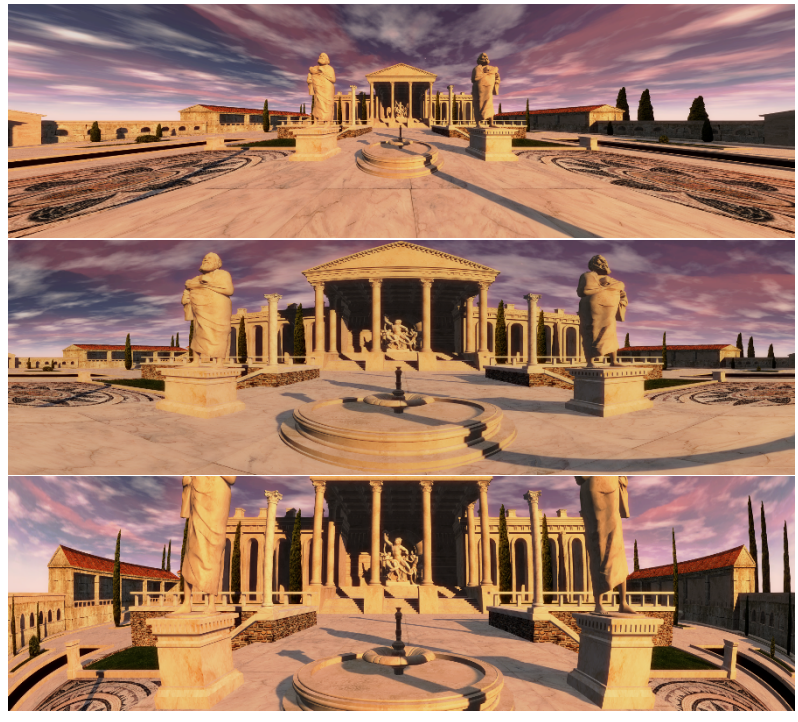


Figure 3-6. *Comparison of projections at a wide 150° horizontal field of view. Top: pinhole perspective. Middle: Panini with d = 1. Bottom: Fisheye. For flythroughs and gaming at a wide field of view, we recommend Panini. Note that the pinhole and Panini projections each have the same horizontal field of view (but different vertical fields of view).*

Listing 3-6. *Generalized Panini projection.*

```
Ray paniniRay(vec2 pixel) {
    vec2 pixelCenterCoords = vec2(pixel) + vec2(0.5f) - (imageSize / 2.f);
    vec2 hv = (pixelCenterCoords / (imageSize.y / 2.f)) * (cameraFovAngle /
         2.f) *
        (bool(cameraFovOrientation == 0) ? 1.f : (float)resolution.y /
             resolution.x);

    float halfFOV = cameraFovAngle / 2.f;
    float halfPaniniFOV = atan(sin(halfFOV), cos(halfFOV) + paniniDistance);

    hv *= halfPaniniFOV;

    float M = sqrt(1 - square(sin(hvPan.x) * paniniDistance)) +
         paniniDistance * cos(hvPan.x);

    float x = sin(hvPan.x) * M;
    float z = (cos(hvPan.x) * M) - paniniDistance;

    float S = (d + 1) / (d + z);

    float y = lerp(tan(hv.y) * (d + z), tan(hv.y) * z, verticalCompression);

    vec3 direction = normalize(x, y, -z);

    return Ray(vec3(0.f), 0.f, direction, INFINITY);
}
```

The generalized Panini projection as implemented in Listing 3-6 is derived as follows. First, the 3D world is projected onto a vertical cylinder of unit radius about the camera. The cylinder is then pinhole projected onto a second camera that is some distance behind the main camera. We call the distance between the two cameras `paniniDistance`, which is any nonnegative number. Increasing this distance interpolates the projection of the cylinder: 0 = rectilinear cylinder projection, 1 = cylindrical stereographic projection. As `paniniDistance` goes to infinity, the projection approaches cylindrical orthographic. The other parameter applies a vertical compression that straightens horizontal lines. We call this parameter `paniniVerticalCompression`, which takes any value from 0 (no compression) to 1 (full compression).

Many photography programs post-process other panoramic formats into Panini projections, and rasterization game engines post-process pinhole projections to Panini. The drawback in each case is that the center of the image will be blurry where it is expanded and the edges will alias and have empty regions where they are compressed. Alternatively, a rasterization

pipeline can perform Panini projection in the vertex shader. Because Panini projection does not preserve all straight lines, this will give incorrect edges and perspective interpolation of attributes such as depth and texture coordinates for large triangles. Ray tracing a Panini projection directly avoids all of these problems.

3.6 FISHEYE

Fisheye projection is a projection of the sphere onto a tangent plane. Unlike pinhole projection, which equalizes distance between pixels, fisheye projection equalizes angular distance. This avoids the skewness of objects in the extreme field of view while introducing significant barrel distortion—straight lines are rendered as curved lines that bow outward from the image center.[2]

As with pinhole projection, fisheye projection is parameterized on the field of view. In addition to horizontal and vertical fields of view, fisheye projections are often specified by a diagonal field of view. This allows a fisheye projection to fully fill a rectangular aspect ratio even at large fields of view.

Code for the fisheye projection is given in Listing 3-7. Images for horizontal, vertical, and diagonal field of view at 180° are given in Figure 3-7.

Listing 3-7. *Fisheye projection.*

```
Ray generateFisheyeRay(vec2 pixel) {
    vec2 clampedHalfFOV = min(cameraFovAngle, pi) / 2.f;
    vec2 angle = (pixel - imageSize / 2.f) * clampedHalfFOV;

    if (cameraFovOrientation == 0) {
        angle = (pixel - imageSize / 2.f) / imageSize.x;
    } else if (cameraFovOrientation == 1) {
        angle = (pixel - imageSize / 2.f) / imageSize.y;
    } else {
        angle = (pixel - imageSize / 2.f) / length(imageSize);
    }

    // Don't generate rays for pixels outside the fisheye
    // (circle and cropped circle only).
    if (length(angle) > 0.5.f * pi) {
        return Ray(vec3(0.f), 0.0f, vec3(0.f), -1);
    }
    vec3 dir = normalize(vec3(sin(angle.x), -sin(angle.y) * cos(angle.x), -
         cos(angle.x) * cos(angle.y)));
    return Ray(vec3(0.f), 0.f, dir, INFINITY);
}
```

[2]The opposite effect, where straight lines bow inward to the image center, is called *pincushion distortion*.



Figure 3-7. *Top: fisheye circle (180° vertical). Middle: cropped circle (180° horizontal). Bottom: full frame (180° diagonal).*

3.7 LENSLET

Lenslet arrays have many applications in near-eye displays and light field rendering [6, 7]. A standard lenslet array is a 2D array of images with the same field of view but different centers of projection. The centers of projection are computed as offsets from the camera origin to simulate a lenslet array camera, which produces lenslet images as in Figure 3-8. The individual microlenses can also be modeled with thin lenses or fisheye lenses if desired.

Real-world near-eye displays often use a specific lens or combination of lenses to create each microlens in the lenslet array. The image formed under each microlens is called the *elemental image*. The display behind the lenses is rendered using a lenslet array projection. Because lenslet arrays require a different center of projection for each microlens, rasterization cannot produce

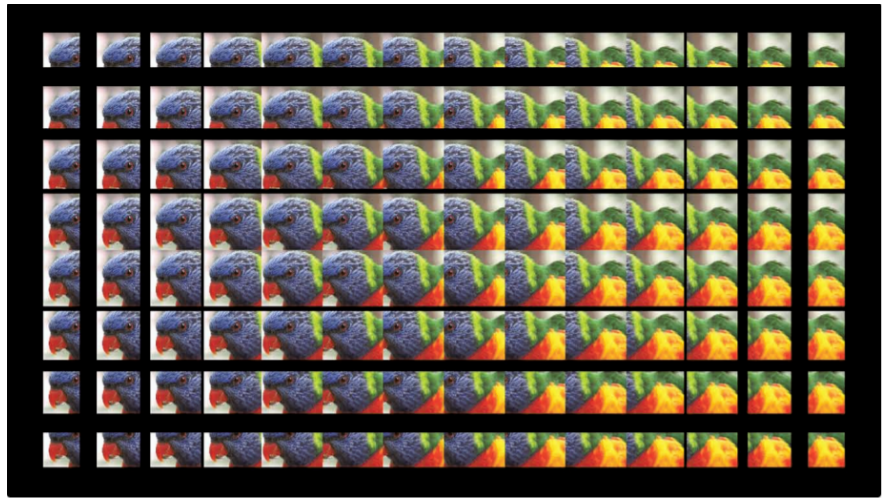


Figure 3-8. *An image through a microlens array in a near-eye display. (Figure from Lanman and Luebke [6].)*

the correct elemental images in a single view—each elemental image must be rasterized separately. Ray tracing the elemental images directly is easily accomplished in a single ray generation shader, which can also provide custom projections or ray patterns depending on the microlens setup.

Figure 3-9 shows a lenslet array with pinhole projection lenses. Ray generation shader code parameterized by the number of lenses and lens separation distance is given in Listing 3-8.



Figure 3-9. *Lenslet array, simulated as a grid of pinhole lenses.*

Listing 3-8. *Parameterized lenslet array with pinhole projection lenses.*

```
Ray generateLensletRay(ivec2 pixel, float numLenses, float lensSeparation) {
    vec2 unitCoord = (pixel.xy - imageSize / 2.f) / length(imageSize);
    float2 lensCenter = round(unitCoord * numLenses) / numLenses;
    float2 lensCoord = (unitCoord - lensCenter) * numLenses;
    vec2 angle = (cameraFovAngle / 2.f) * lensCoord;
    vec3 dir   = normalize(tan(angle.x), tan(angle.y), -1);
    return Ray(lensCenter * lensSeparation, 0.f, dir, inf);
}
```

3.8 OCTAHEDRAL

Spherical to square projections are useful for sky domes, reflection probes, and light field probes. There are many such projections, including the classic graphics sphere maps and cube maps. As shown in Figure 3-10, an octahedral mapping [9] of the sphere produces relatively low distortion, has few boundary edges, and maps the entire sphere to a single square. These properties make it a popular modern choice.

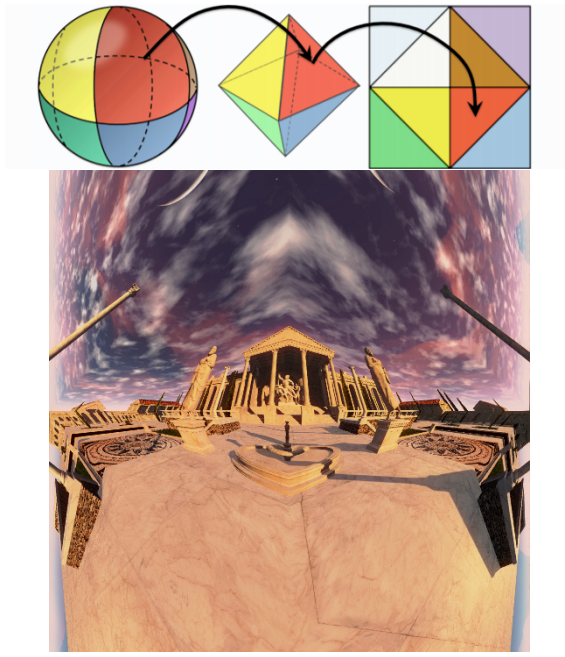


Figure 3-10. *Partitioning of the sphere into an octahedron flattened to the plane. (Diagram from Cigolle et al. [3].)*

Listing 3-9. *Pixel centers in a square image mapped to octahedral rays on the sphere.*

```
float signNotZero(in float k) {
    return (k >= 0.f) ? 1.f : -1.f;
}

vec2 signNotZero(in vec2 v) {
    return vec2(signNotZero(v.x), signNotZero(v.y));
}

vec2 octEncode(vec3 v) {
    float l1norm = abs(v.x) + abs(v.y) + abs(v.z);
    vec2 result = v.xy * (1.f /  l1norm);
    if (v.z < 0.f) {
        result = (1.f - abs(result.yx)) * signNotZero(result.xy);
    }
    return result;
}

vec3 octDecode(vec2 o) {
    vec3 v = vec3(o.x, o.y, 1.f - abs(o.x) - abs(o.y));
    if (v.z < 0.f) {
        v.xy = (1.f - abs(v.yx)) * signNotZero(v.xy);
    }
    return normalize(v);
}

vec3 generateOctahedralRay(vec2 pixel) {
    pixel /= imageSize; // Image is square for octahedral projection.
    pixel = (pixel - vec2(0.5f)) * 2.f;
    return Ray(vec3(0), octDecode(pixel), 0.f, INFINITY);
}
```

The projection operates by folding each corner of the square into the center, thus double-covering a diamond shape. Inflating this while maintaining edges and faces produces an octahedron, the eight-sided regular polyhedron, with no distortion. Further inflating the octahedron without preserving edges or faces maps that shape to the full sphere with minimal distortion. Pixel adjacency on the interior of the original square is the same as adjacency on the sphere, and for each half-edge of the square, there is a simple adjacency mapping to the reflected half-edge. All four corners of the square map to a single point on the sphere.

Listing 3-9 takes each texel center in the image and projects it through the octahedral mapping into a ray direction. For convenience, the inverse mapping from direction to pixel is also shown.

Listing 3-10. *Per-face six-texture projection of a cube map.*

```
vec3 generateCubeMapRay(vec2 pixel, int face, int API) {
    // Create the ray in the space of the cube map face.
    Ray cubemapSpaceRay = pinholeRay(pixel);

    mat3x3 faceRotation = { 1.f, 0.f, 0.f,
                            0.f, 1.f, 0.f,
                            0.f, 0.f,-1.f };
    switch(face) {
    case 0: // +x
       faceRotation = { 0.f, 0.f,-1.f,
                        0.f, 1.f, 0.f,
                        1.f, 0.f, 0.f };
    case 1: // -x
       faceRotation = { 0.f, 0.f, 1.f,
                        0.f, 1.f, 0.f,
                       -1.f, 0.f, 0.f };
    case 2: // +y
       faceRotation = {-1.f, 0.f, 0.f,
                        0.f, 0.f,-1.f,
                        0.f,-1.f, 0.f };
    case 3: // -y
       faceRotation = {-1.f, 0.f, 0.f,
                        0.f, 0.f, 1.f,
                        0.f, 1.f, 0.f };
    case 4: // +z
       faceRotation = {-1.f, 0.f, 0.f,
                        0.f, 1.f, 0.f,
                        0.f, 0.f,-1.f };
    case 5: // -z
        // Nothing to do
    }

    return vec3(0, faceRotation * cubemapSpaceRay.direction, 0, inf);
}
```

3.9 CUBE MAP

Cube maps project a sphere onto the six faces of a cube. Each face of a cube map is just a 90° field of view, square image looking along a different axis, so the rays for rendering a cube map are generated by the pinhole camera projection. However, there are many conventions for the ordering and orientation of the faces. Listing 3-10 gives the DirectX, Vulkan, and OpenGL transformation matrices for each face for use with pinhole projection.

Figure 3-11 shows a common alternative visualization for a cube map in which all six faces are packed into a single rectangular texture as a cross. This atlas is inefficient at runtime for memory and for sampling because it is not supported by hardware cube map sampling and filtering. However, it is

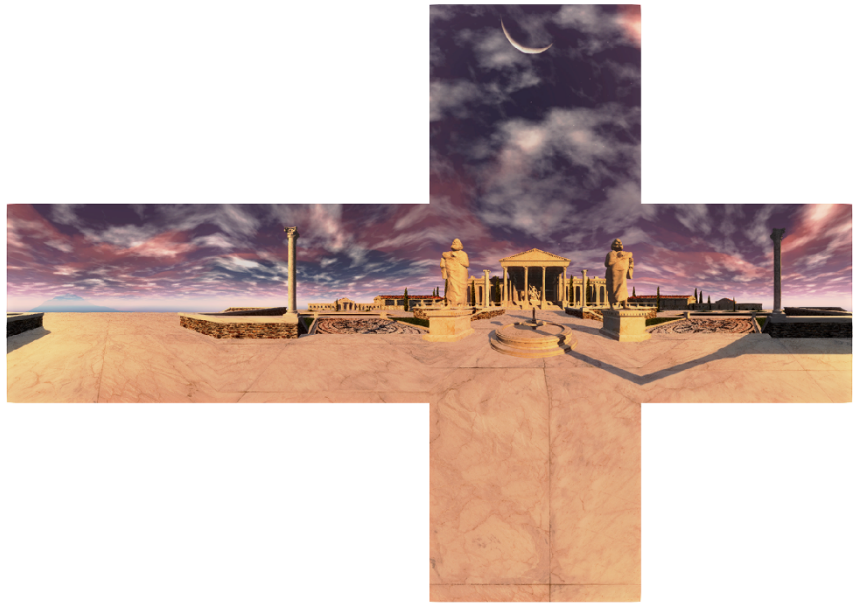


Figure 3-11. *Six faces of a cube map visualized as a cross.*

useful for visualization and artist manipulation, and some engines accept cube maps in this format directly.

3.10 ORTHOGRAPHIC

Orthographic projections are common for engineering and scientific visualization applications and were once common in 2.5D video games. All camera rays in an orthographic projection are parallel, so there is no perspective scale change. This is also the projection used by shadow maps for directional lights. As can be seen in Figure 3-12, orthographic projection preserves straight lines, but not lengths or angles.

An orthographic camera is parameterized by the world-space extent of the image and aperture because it has no perspective; the field of view angle is effectively zero degrees. Listing 3-11 gives the orthographic ray generation shader.



Figure 3-12. *Orthographic projection.*

Listing 3-11. *Orthographic projection, where the field of view is in distance units instead of radians.*

```
vec3 generateOrthographicRay(vec2 pixel) {
    pixel *= cameraFovDistance;
    vec3 origin = vec3(pixel, 0.f);
    return Ray(origin, vec3(0.f,0.f,-1.f), 0.f, INFINITY);
}
```

3.11 FIBONACCI SPHERE

Algorithms such as ambient occlusion and radiosity sample rays in a sphere or hemisphere and then weight and filter the results. Latitude-longitude generation produces rays that bunch up at the poles and are spread out near the equator, as well as aligning on great circle arcs in ways that produce poor sampling patterns. A better way to generate uniformly distributed rays on a sphere is the spherical Fibonacci [5] pattern shown in Figure 3-13.

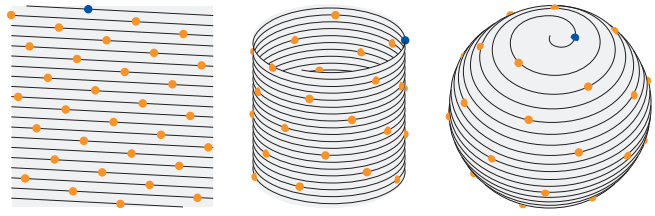


Figure 3-13. *The spherical Fibonacci pattern (right) distributes ray directions uniformly and with low discrepancy by wrapping a rotated regular lattice around a cylinder (center) and then projecting onto a sphere (left). (Figure from Keinert et al. [5].)*

Listing 3-12. *Uniform, low-discrepancy rays in all directions using spherical Fibonacci mapping.*

```
vec3 sphericalFibonacci(float i, float n) {
    const float PHI = sqrt(5) * 0.5f + 0.5f;
    float fraction = (i * (PHI - 1)) - floor(i * (PHI - 1));
    float phi = 2.f * pi * fraction;
    float cosTheta = 1.f - (2.f * i + 1.f) * (1.f / n);
    float sinTheta = sqrt(saturate(1.f - cosTheta*cosTheta));

    return vec3(cos(phi) * sinTheta, sin(phi) * sinTheta, cosTheta);
}

Ray generateSphericalFibonacciRay(int index) {
    float numRays = imageSize.x * imageSize.y;
    return Ray(vec3(0.f), 0.f, sphericalFibonacci(index, numRays), inf);
}
```

Listing 3-12 gives the code for generating the *i*th of *n* Fibonacci points on a sphere, which the ray generation shader then uses to produce the ray direction. In this case, the "image" doesn't make sense in a 2D context, but it is often still a good way to store the ray hits, G-buffer values, and so on, so we choose `n` to be the product of the image dimensions.

REFERENCES

[1] Barsky, B., Horn, D., Klein, S., Pang, J., and Yu, M. Camera models and optical systems used in computer graphics: Part I, object-based techniques. In *Computational Science and Its Applications—ICCSA 2003*, volume 2669 of *Lecture Notes in Computer Science*, pages 246–255, May 2003. DOI: 10.1007/3-540-44842-X_26.

[2] Blinn, J. F. A trip down the graphics pipeline: The homogeneous perspective transform. *IEEE Computer Graphics and Applications*, 13(3):75–80, 1993. DOI: 10.1109/38.210494.

[3] Cigolle, Z. H., Donow, S., Evangelakos, D., Mara, M., McGuire, M., and Meyer, Q. A survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques (JCGT)*, 3(2):1–30, 2014. http://jcgt.org/published/0003/02/01/.

[4] Jimenez, J., Gutierrez, D., Yang, J., Reshetov, A., Demoreuille, P., Berghoff, T., Perthuis, C., Yu, H., McGuire, M., Lottes, T., Malan, H., Persson, E., Andreev, D., and Sousa, T. Filtering approaches for real-time anti-aliasing. In *ACM SIGGRAPH 2011 Courses*, 6:1–6:329, 2011. DOI: 10.1145/2037636.2037642.

[5] Keinert, B., Innmann, M., Sänger, M., and Stamminger, M. Spherical Fibonacci mapping. *ACM Transactions on Graphics*, 34:193:1–193:7, Oct. 2015. DOI: 10.1145/2816795.2818131.

[6] Lanman, D. and Luebke, D. Near-eye light field displays. *ACM Transactions on Graphics*, 32(6):220:1–220:10, 2013. DOI: 10.1145/2508363.2508366.

[7] Levoy, M. and Hanrahan, P. Light field rendering. In *Proceedings of SIGGRAPH '96*, pages 31–42, 1996. DOI: 10.1145/237170.237199.

[8] McGuire, M. *The Graphics Codex*. Casual Effects, 2.14 edition, 2018. http://graphicscodex.com.

[9] Meyer, Q., Süßmuth, J., Sußner, G., Stamminger, M., and Greiner, G. On floating-point normal vectors. *Computer Graphics Forum*, 29(4):1405–1409, 2010. DOI: https://doi.org/10.1111/j.1467-8659.2010.01737.x.

[10] Microsoft Documentation. D3d11 standard multisample quality levels. https://docs.microsoft.com/en-us/windows/win32/api/d3d11/ne-d3d11-d3d11_standard_multisample_quality_levels, 2018.

[11] Sharpless, T. K., Postle, B., and German, D. M. Pannini: A new projection for rendering wide angle perspective images. In *Computational Aesthetics '10*, pages 9–16, 2010.

[12] Yang, L., Liu, S., and Salvi, M. A survey of temporal antialiasing techniques. *Computer Graphics Forum*, 39(2):607–621, 2020. DOI: 10.1111/cgf.14018.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 4

HACKING THE SHADOW TERMINATOR

Johannes Hanika
KIT/Weta Digital

ABSTRACT

Using ray tracing for shadows is a great method to get accurate direct illumination: precise hard shadows throughout the scene at all scales, and beautiful penumbras from soft box light sources. However, there is a long-standing and well-known issue: the *terminator problem*. Low tessellation rates are often necessary to reduce the overall load during rendering, especially when dynamic geometry forces us to rebuild a bounding volume hierarchy for ray tracing every frame. Such coarse tessellation is often compensated by using smooth shading, employing interpolated *vertex normals*. This mismatch between geometric normal and shading normal causes various issues for light transport simulation. One is that the geometric shadow, as very accurately reproduced by ray tracing (see Figure 4-1, left), does in fact not resemble the smooth rendition we are looking for (see Figure 4-1, right). This chapter reviews and analyzes a simple hack-style solution to the terminator problem. Analogously to using shading

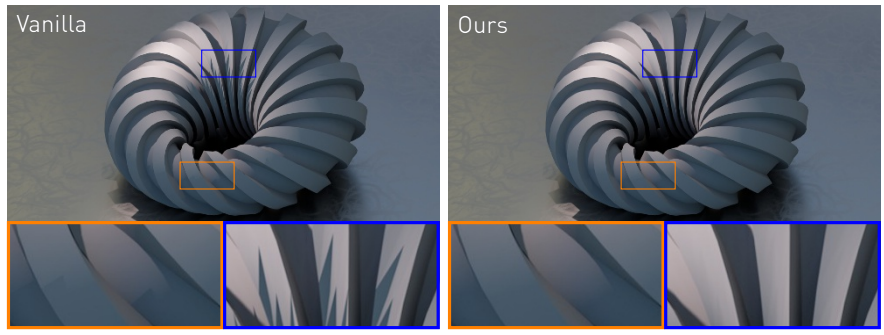


Figure 4-1. *Low-polygon ray tracing renders are economical with respect to acceleration structure build times, but introduce objectionable artifacts with shadow rays. This is especially apparent with intricate geometry such as this twisted shape (left). In this chapter, we examine a simple and efficient hack to alleviate these issues (right).*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_4

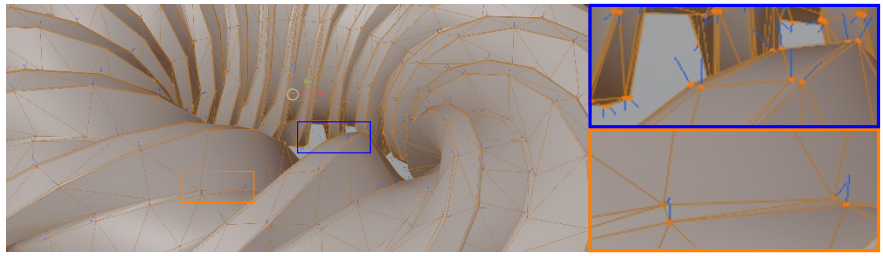


Figure 4-2. *A screen capture of the Blender viewport, to visualize the mesh and vertex normals used for Figure 4-1. The triangles come from twisted quads and have extremely varying normals. There are tiny and elongated triangles at the bevel borders especially at the back, which is where some artifacts remain in the render.*

normals that are smooth but inconsistent with the geometry, we will use shading points which are inconsistent with the geometry but result in smooth shadows. We show how this is closely related to quadratic Bézier surfaces but cheaper and more robust to evaluate.

4.1 INTRODUCTION

Ray tracing has been an elegant and versatile method to render 3D imagery for the better part of the last 50 years [1]. There has been a constant push to improve the performance of the technique throughout the years. Recently, we have seen dedicated ray tracing hardware units that even make this approach viable for real-time applications. However, since this operates hard at the boundary of the possible, some classic problems resurface. In particular, issues with low geometric complexity and simple and fast approximations are strikingly similar to issues that the community worked on in the 1980s and 1990s. In this chapter we discuss one of these: *the terminator problem*.

In 3D rendering, geometry is often represented as a polygon mesh. In fact, today triangle meshes are the ubiquitous choice. While quad meshes are often used to reduce memory footprint, individual quads are often treated as two triangles (instead of a bilinear patch) internally. We will thus limit our discussion here to triangle meshes. These can be used for intricate shapes such as Figure 4-1, which are tessellated and triangulated as illustrated in Figure 4-2. In this particular example, the shape is only very coarsely tessellated. The mesh contains long and thin triangles as well as a large variation of normals throughout one triangle (marked with blue in the image).

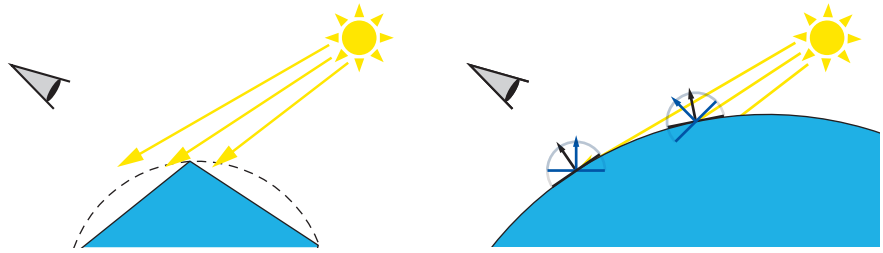


Figure 4-3. *Two incarnations of the terminator problem. Left: a (very) coarsely tessellated sphere is illuminated. The sphere surface (dashed) we were trying to approximate would show smooth shadow falloff, but the flat triangle surface facing the eye in this example would be rendered completely black. Using vertex normals (i.e., Phong shading) to evaluate the materials smoothly does unfortunately not render the shadow rays visible. For this to happen, we need to start the shadow rays at the dashed surface. This chapter proposes a simple and cheap way to do this. Right: the bump terminator problem is caused by mismatching hemispheres defined by the geometric normals (black) and the shading normals (blue). This kind of problem persists even when using smooth base geometry.*

To hide the discretization artifacts coming from this, shading is traditionally performed using a normal resulting from barycentric interpolation of *vertex normals* across the triangle [9]. Using normals that are inconsistent with the geometric surface normal can lead to various issues, for instance with symmetry in the light transport operator [13]. Woo et al. [16] point out a particular issue with ray traced shadows, which we illustrate again in Figure 4-3, left. Say we want to render the dashed, smooth surface, but an accurate representation is too expensive to intersect with a ray. We thus use a triangle mesh indicated by the blue solid. The triangle facing the viewpoint to the left should show a smooth falloff in shading to approximate the dashed surface well. Instead, because it is facing away from the light source, it will be rendered completely black: rays traced from the triangle surface to the light will correctly and accurately report that this surface is in shadow.

This problem is well understood and solutions using small user-driven epsilon values to push out the shading point from the triangle surface have been proposed as early as 1987 [11]. Some years later, CPU ray tracing had advanced significantly [15, 2], so objects with low tessellation became interesting for real-time display. Such meshes are prone to showing the terminator problem. Consequently, in a side note in [7, Section 5.2.9], a simple way of determining an adaptive epsilon value was proposed as an inexpensive workaround. In this chapter, we evaluate this approach in a bit more depth.

4.2 RELATED WORK

Over the years, many approaches have been proposed to work around the problems arising with shading normals. They can be roughly classified into three groups. The first deals with geometry terms arising in bidirectional light transport. Veach [13] observed that shading normals introduce an inconsistency between path tracing and light tracing. He proposed a correction factor based on the ratio of cosines between the geometry and shading normals. The pictures in this chapter are rendered with a unidirectional path tracer and thus do not use this correction factor. In fact, the method proposed here, as is mostly the case when working with shading normals, is not reciprocal.

The second class involves smoothing the shadow terminator by altering the shading of a microfacet material model. These approaches start from the observation that a shading normal other than the geometric normal can be pushed inside the microfacet model, as an off-center normal distribution. This idea can be turned into a consistent microsurface model [10], and from there the surface reflectance can be derived by first principles. Since the extra roughness introduced into the microsurface will lead to some overshadowing, multiple scattering between microfacets can be taken into account to brighten the look. This technique has been simplified for better adoption in practice and refined to reduce artifacts caused by the simplifications [3, 4, 5]. These approaches start from the observation that a bump map can make the reflectance extend too far into the region where the geometric normal is, in fact, shadowed (see the left two mismatching hemispheres in Figure 4-3). In this case the result will be black, but the material response is still bright, leading to a harsh shading discontinuity. The simple solution provided is to introduce a shadow falloff term that makes sure that the material evaluation smoothly fades to black. This is illustrated in Figure 4-4. As an example, we implemented Conty et al.'s method [4]. Note that our material is diffuse, and thus violates the assumptions they made about how much energy of the lobe is captured in a certain angular range. Thus, the method moves the shadow region, but not far enough by a large margin, at least for this very coarsely tessellated geometry. The best we could hope for here is to darken the gradient so much that it hides the coarse triangles completely, leading to significant look changes as compared to the base version.

In some sense Keller at al. [8, Figure 21] are also changing the material model. They bend the shading normal depending on the incoming ray, such

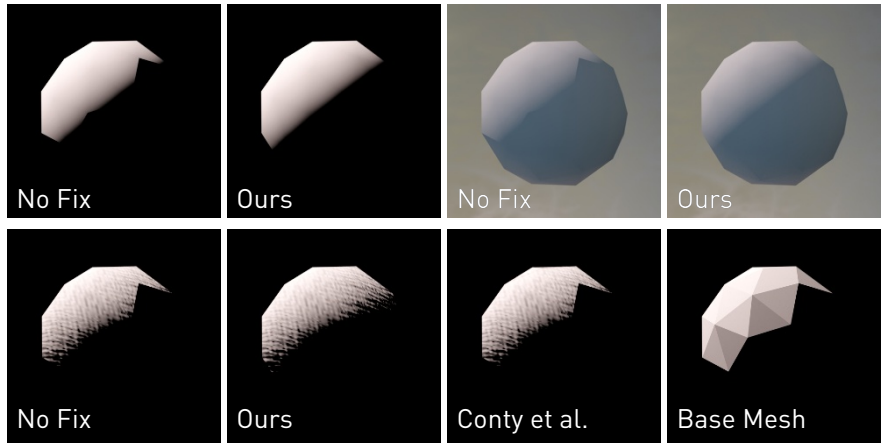


Figure 4-4. *This is how the problem illustrated in Figure 4-3 manifests itself in practice. Top row: without bump map; left two: direct illumination from a spotlight only; right two: with global illumination and an environment map. Note that the indirect lighting at the bottom of the sphere does not push out the shading point in any of the images. Bottom row: with a bump map applied. The hack leaves the look of the surface untouched as much as possible. Conty et al.'s shadowing term [4] does not help on the diffuse surface.*

that a perfectly reflected ray would still be just above the geometric surface. This changes the hemisphere of the shading normal, whereas in a sense we change the hemisphere of the geometric normal. At least we make sure that some shadow rays will yield a nonzero result even when cast under the geometric surface.

In general, it is hard to create a consistent microsurface model in the presence of both normal maps and vertex normals. Figure 4-5 illustrates the issue. Schüßler et al. [10] cut the surface into Fresnel lens–like microsteps, where one side (orange in the figure) corresponds to the shading normal. To complete the model and make it physically consistent, there needs to be another microfacet orientation (drawn in light blue) to close the surface. When two triangles meet at the same vertex with the same normal, the orientation of these additional microfacets lead to a discontinuity.

On the other hand, we only want to smooth out the geometric shadow terminator; i.e., we are dealing with shadow rays more than with misaligned hemispheres for geometric and shading normal. This means that we can make assumptions about slow and smooth variation of our normals across the whole triangle. It follows that the technique examined here does not work

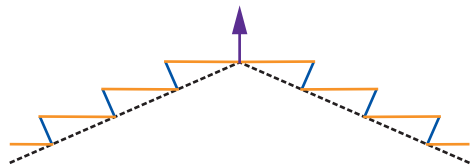


Figure 4-5. *The microsurface model of Schüßler et al. [10] at a vertex with a vertex normal (blue). The facets oriented toward the shading normal (orange) have a smooth transition at the vertex (facing the same direction in this closeup). However, the light blue facets orthogonal to the geometric surface (dashed) will create a discontinuous look.*

for normal maps, but could likely be combined with a microfacet model addressing the hemisphere problem.

The third approach is the obvious choice: resolve issues with coarse tessellation by tessellating more finely. A closely related technique is called *PN triangles* [14]. One constructs Bézier patches from vertices and normals, usually the cubic version [12]. Van Overveld and Wyvill [12] also mention the possibility to do quadratic, which is more closely related to the technique examined here. The main difference is that we don't want to tessellate but only fix the shadows instead.

4.3 MOVING THE INTERSECTION POINT IN HINDSIGHT

As a minimally invasive change to fix the harsh shadow from coarsely tessellated geometry, we want to move only the primary intersection point away from the triangle, ideally to a location on a smooth freeform surface (i.e., the dashed surface in Figure 4-3). For this, we want to look at how a simple quadratic Bézier patch is constructed from vertices and vertex normals.

Figure 4-6 shows a triangle with vertices A, B, C, an intersection point P, and an illustration of the barycentric coordinates u, v, w on the left. To construct a point P' on a quadratic Bézier patch defined by these vertices, these barycentric coordinates, and the vertex normals n_A, n_B, n_C, we use de Casteljau's algorithm. First, we construct additional control points AB, BC, CA. We have some freedom in how to do this; options have been discussed in the literature [14, 12]. Then, we use the barycentric coordinates u, v, w to compute three additional vertices A', B', and C' from the three triangles (A, AB, CA), (AB, B, BC), and (CA, BC, C), respectively. These three new points form another triangle, which we interpolate once more with the barycentric coordinates u, v, w to finally arrive at P'.

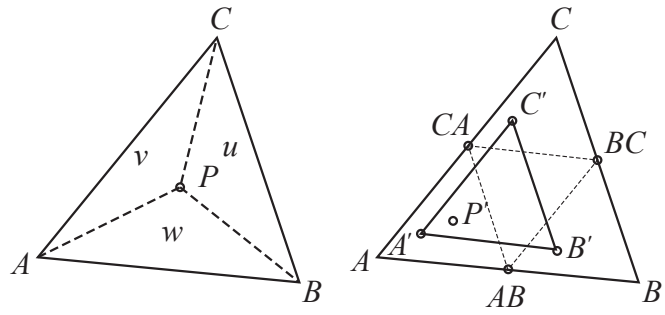


Figure 4-6. *Left: illustration to show our naming scheme inside a triangle. Right: schematic of the barycentric version of de Casteljau's algorithm. P' is computed as the result of a quadratic barycentric Bézier patch, defined by the corner points A, B, C as well as the extra nodes AB, BC, CA. These will in general not lie in the plane of the triangle ABC.*

Let's have a look how to place the extra control points such as AB. The patches should not have cracks between them, so the placement can only depend on the data available on the edge, for instance, A, B, n_A, n_B. Note that, even then, using a real Bézier patch as geometry would potentially open cracks at creases where a mesh defines different normals for the same vertex on different faces. When only moving the starting point of the shadow ray, this is not a problem. We could, for instance, place AB at the intersection of three planes: the two tangent planes at A, n_A and B, n_B as well as the half-plane at $(A + B)/2$ with a normal in the direction of the edge $B - A$. These three conditions result in a 3×3 linear system of equations to solve. This requires a bit of computation, and there is also the possibility that there is no unique solution.

Instead, let's look at the simple technique proposed in [7, Section 5.2.9]. The procedure is similar in spirit to a quadratic Bézier patch as we just discussed it, and it is summarized in pseudocode in Listing 4-1. An intermediate triangle is constructed, and the final point P' is placed in it by interpolation using the barycentric coordinates u, v, w. The difference to the quadratic patch is the way this triangle is constructed (named `tmpu, tmpv, tmpw` in the code listing). To avoid the need for the extra control points on each edge, the algorithm proceeds as follows: the vector from each corner of the triangle to the flat intersection point P is computed and subsequently projected onto the tangent plane at this corner. This is illustrated in Figure 4-7, right. This procedure is very simple and efficient, but comes with a few properties that are different than a real Bézier surface, which we will discuss next.

Listing 4-1. *Pseudocode of the simple shading point offset procedure outlined in [7].*

```
// get distance vectors from triangle vertices
vec3 tmpu = P - A, tmpv = P - B, tmpw = P - C
// project these onto the tangent planes
// defined by the shading normals
float dotu = min(0.0, dot(tmpu, nA))
float dotv = min(0.0, dot(tmpv, nB))
float dotw = min(0.0, dot(tmpw, nC))
tmpu -= dotu*nA
tmpv -= dotv*nB
tmpw -= dotw*nC
// finally P' is the barycentric mean of these three
vec3 Pp = P + u*tmpu + v*tmpv + w*tmpw
```

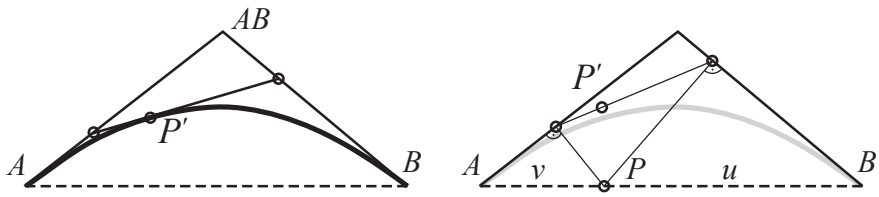


Figure 4-7. *Left: 2D side view visualization of a quadratic Bézier patch (for the w = 0 case), showing the line between A and B. The barycentric coordinates (u, v) are relevant for this 2D slice: de Casteljau proceeds by interpolating A and AB in the ratio v:u, as well as AB and B. The two results are then interpolated using v:u again to yield P′. Right: our simple version does not require the point AB for the computation: it projects the intersection point P on the flat triangle orthogonally to the tangent planes at A and B (note that AB lies on both of these planes, but we don't need to know where). These temporary points are then interpolated in the ratio v:u again, to yield the shading intersection point P′. The Bézier line from the left is replicated to emphasize the differences.*

4.4 ANALYSIS

To evaluate the behavior of our cheap approximation, we plotted a few side views in flatland, comparing a quadratic Bézier surface to the surface resulting from the pseudocode in Listing 4-1. The results can be seen in Figure 4-8.

The first row shows a few canonical cases with distinct differences between the two approaches. In the first case in Figure 4-8a, the two surfaces are identical, as both vertex normals point outward with a 45° angle to the geometric normal. In Figure 4-8b, an asymmetric case is shown; note how the point *AB* is moved toward the left. It can be seen that our surface (green) has more displacement from the flat triangle, only approximately follows the

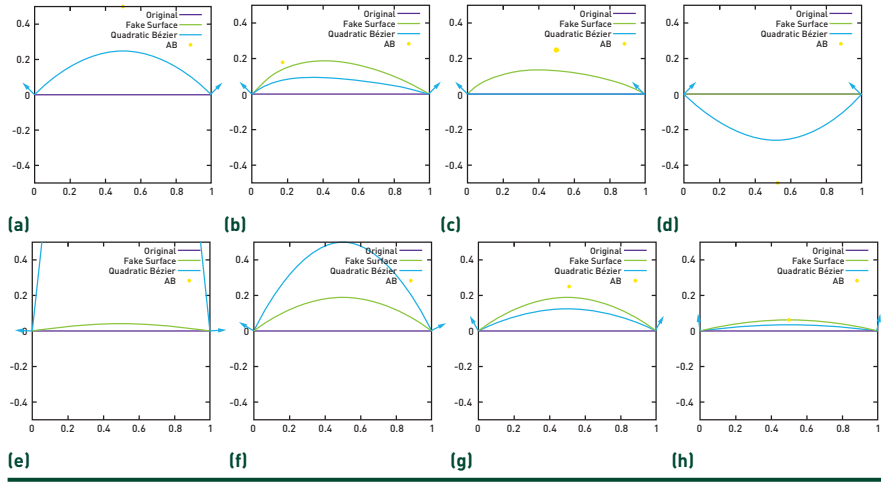


Figure 4-8. *2D comparison against quadratic Bézier: (a) Canonical case, equivalent. (b) Asymmetric case, no tangent at B and out of convex hull with control point AB. (c) Degenerate case not captured by quadratic Bézier. (d) Concave case clamped by `min()` in our code. (e–h) The behavior when moving AB toward the geometric surface.*

tangent planes at both vertices, and is placed outside the convex hull of the control cage of the Bézier curve. Though all this may be severe downsides for sophisticated applications, the deviations from the Bézier behavior may be acceptable for a simple offset of the shadow ray origin.

In Figure 4-8c, a degenerate case is shown: both vertex normals point in the same direction. This happens, for instance, in the shape in Figure 4-2 at the flanks of the elevated structures, where one normal is interpolated toward the top and one toward the bottom, but with both pointing the same direction. The approach to solve for *AB* by plane intersection now fails because the two tangent planes are parallel, and no quadratic Bézier can be constructed. This is a robustness concern, especially if the vertex normals aren't specifically authored for quadratic patches or are animated.

In Figure 4-8d the concave case is shown. Because we clamp away positive dot products, such a case will evaluate to the flat surface instead of bulging to the inside of the shape. This is the desired behavior: we don't push shadow ray origins inside the object.

The second row varies the distance of *AB* to the surface. In Figures 4-8e and 4-8f, *AB* is far away from the surface, where the former is almost degenerate. The quadratic Bézier patch consequently moves the curve very far away, too. This may not be the desired behavior in our case, as animation

might have caused a normal to be this extreme by accident. The cheap approximation stays much closer to the geometric surface.

Figures 4-8g and 4-8h show vertex normals closer to the geometric normal than 45°. Here, the situation turns around, and the cheap surface is farther away from the geometry than the Bézier patch. At these small deviations, however, this is much less of a concern.

In summary, the cheap surface is only approximately tangent at the vertices and violates the convex hull property with respect to the control cage.

4.5 DISCUSSION AND LIMITATIONS

The surface examined in this chapter does not strictly follow the tangent condition at the vertices. Because we only use it to offset the origin of the shadow ray, this is not a big issue: the shading itself will use the vertex normal to determine the upper hemisphere for lighting.

We have seen that the cheap surface does not obey the control cage of the Bézier patch. Instead, it is farther from the surface for small vertex normal variations, and closer for large variations, as compared to the Bézier surface. As the interpolation is just quadratic, it does not overshoot or introduce ringing artifacts of any kind.

The larger offsets for small variations may lead to shadow boundaries that are slightly more wobbly than expected. See, for instance, Figure 4-1 just above the blue inset. The shadow edge was already choppy because of the low tessellation both in the shadow caster and in the receiver. Offsetting the shadow ray origin seemed to aggravate this issue. However, in the foreground (to the left of the orange inset), the opposite can be observed: the shadow edges look smoother than before.

We have seen that the surface offset can work with concave objects in a sense that it will not push the shadow ray origins inside the object. However, special care has to be taken when working with transparent objects. Transmitted rays may need to apply the offset the other way around, i.e., flip all the vertex normals before evaluation.

The mesh in Figure 4-2 was modeled with bevel borders, i.e., the sharp corners consist of an extra row of small polygons to make sure that the edge appears sharp. If such creases are instead modeled using face-varying vertex normals, the shadow ray origins will have a discontinuous break at the edge.

Note that the surface will still be closed and the shading will depend on the normals, so this only affects the shadows.

Further, the method is specifically tailored for vertex normals. This means that for multi-lobe and off-center microfacet models, such as multi-modal LEADR maps [6], there is still a necessity to adapt microfacet surface models or adjust the shadowing and masking terms.

We only discussed shadow rays for evaluation of direct lighting. The same can be said about starting indirect lights when material scattering is used. In this case, the effect of the shadow terminator is a bit different and more subtle; see, for instance, the bottom half of the spheres with global illumination enabled in Figure 4-4.

4.6 CONCLUSION

We discussed a simple and inexpensive side note on the terminator problem from the time when CPU ray tracing became interesting for real-time applications. This method effectively resembles quadratic Bézier patches, but is cheaper to evaluate and also works in degenerate cases where creating the additional control points for such a patch would be ill-posed. It only offsets the shading point from which the shadow ray is cast; the surface itself remains unchanged. This means that low-polygon meshes will receive smooth shadows without resorting to more heavyweight tessellation approaches that may require a bounding volume hierarchy rebuild, too. In the long run, the problems of low-polygon meshes may go away as finer tessellations will become viable for real-time ray tracing. Until then, this technique has a valid use case again.

REFERENCES

[1] Appel, A. Some techniques for shading machine renderings of solids. In *American Federation of Information Processing Societies*, volume 32 of *AFIPS Conference Proceedings*, pages 37–45, 1968. DOI: 10.1145/1468075.1468082.

[2] Benthin, C. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, Jan. 2006.

[3] Chiang, M. J.-Y., Li, Y. K., and Burley, B. Taming the shadow terminator. In *ACM SIGGRAPH 2019 Talks*, 71:1–71:2, 2019. DOI: 10.1145/3306307.3328172.

[4] Conty Estevez, A., Lecocq, P., and Stein, C. A microfacet-based shadowing function to solve the bump terminator problem. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 12. Apress, 2019.

[5] Deshmukh, P. and Green, B. Predictable and targeted softening of the shadow terminator. In *ACM SIGGRAPH 2020 Talks*, 6:1–6:2, 2020. DOI: 10.1145/3388767.3407371.

[6] Dupuy, J., Heitz, E., Iehl, J.-C., Poulin, P., Neyret, F., and Ostromoukhov, V. Linear efficient antialiased displacement and reflectance mapping. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 32(6):Article No. 211, 2013. DOI: 10.1145/2508363.2508422.

[7] Hanika, J. *Spectral Light Transport Simulation Using a Precision-Based Ray Tracing Architecture*. PhD thesis, Ulm University, 2011.

[8] Keller, A., Wächter, C., Raab, M., Seibert, D., van Antwerpen, D., Korndörfer, J., and Kettner, L. The iray light transport simulation and rendering system. In *ACM SIGGRAPH 2017 Talks*, 34:1–34:2, 2017. DOI: 10.1145/3084363.3085050.

[9] Phong, B. T. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975. DOI: 10.1145/360825.360839.

[10] Schüßler, V., Heitz, E., Hanika, J., and Dachsbacher, C. Microfacet-based normal mapping for robust Monte Carlo path tracing. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 36(6):205:1–205:12, Nov. 2017. DOI: 10.1145/3130800.3130806.

[11] Snyder, J. and Barr, A. Ray tracing complex models containing surface tessellations. *Computer Graphics*, 21(4):119–128, 1987.

[12] Van Overveld, C. and Wyvill, B. An algorithm for polygon subdivision based on vertex normals. In *Proceedings of Computer Graphics International Conference*, pages 3–12, Jan. 1997. DOI: 10.1109/CGI.1997.601259.

[13] Veach, E. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1998.

[14] Vlachos, A., Peters, J., Boyd, C., and Mitchell, J. L. Curved PN triangles. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pages 159–166, 2001. DOI: 10.1145/364338.364387.

[15] Wald, I. and Slusallek, P. State of the art in interactive ray tracing. In *Eurographics 2001—STARs*, 2001. DOI: 10.2312/egst.20011050.

[16] Woo, A., Pearce, A., and Ouellette, M. It's really not a rendering bug, you see ... *IEEE Computer Graphics and Applications*, 16(5):21–26, Sept. 1996. DOI: 10.1109/38.536271.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 5

SAMPLING TEXTURES WITH MISSING DERIVATIVES

Maksim Aizenshtein and Matt Pharr
NVIDIA

ABSTRACT

A key part of rendering high-quality imagery without aliasing has been to use first-order differentials of texture coordinates to determine spatially varying filter kernels for texture map lookups. With traditional rasterization, a variety of techniques have been developed to do this, but the most common one used today is to compute differences among adjacent pixels in a 2×2 quad of pixels. However, doing so is not always possible with modern rendering techniques, including both some forms of deferred shading as well as ray tracing.

Approaches based on ray differentials and ray cones have successfully been applied to this problem in ray tracing, but it is also possible to compute these differentials analytically based on the projective transformation and the scene geometry. This chapter derives such an approach, which requires no auxiliary storage, is computationally efficient, and requires only a few tens of lines of code to implement. An implementation of the technique is included in the supplementary material.

5.1 INTRODUCTION

Modern GPU pixel shaders provide explicit derivative functions, `ddx()` and `ddy()` in HLSL and `dFdx()` and `dFdy()` in GLSL. These functions are capable of estimating derivatives of any expression and do so by computing first-order differences with adjacent pixels. When texture lookup functions are used in shaders, such derivatives are computed automatically by the GPU based on the provided texture coordinates in order to both determine which mip level to use as well as what area to filter over in that mip level.

With modern rendering techniques, those functions may be either not available or not applicable. For example, with deferred shading, if it is necessary to perform texture lookups after the G-buffer has been created, then although taking differences of texture coordinates with adjacent pixels

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_5

gives a reasonable result if the adjacent pixel is part of the same primitive, it may give wildly incorrect results if it is not. When the texture is fetched during rasterization, the GPU takes care of this edge case (so to speak), so it is not a concern. Though texture coordinate derivatives could be stored as additional channels in the G-buffer for this case, doing so would consume both memory and bandwidth.

Another important case is ray tracing. Methods like ray differentials [5] have been applied to carry derivative information along with rays, giving accurate texture coordinate derivative estimates that lead to well antialiased images, though at a relatively high storage cost. Especially on the GPU, where keeping the size of the ray payload small is important for performance, this cost may be prohibitive. Akenine-Möller et al. [1, 2] have developed more lightweight ray derivative representations that give good results. Both of these approaches can be applied to computing texture derivatives on geometry that is directly visible from the camera without using any auxiliary storage.

The method described in this chapter provides another option. It directly computes derivatives at any point in the scene that are fully accurate for points directly visible from the camera, without requiring any additional storage. Unlike ray differentials and ray cones, however, it does not account for the effect of reflection or refraction from surfaces in the filter estimates that it computes for secondary ray intersection points; if curved specular surfaces are present in the scene, this approach may lead to either aliasing or blurring for texture lookups at secondary reflections.

5.2 TEXTURE COORDINATE DERIVATIVES AT VISIBLE POINTS

For a point on a triangle that is inside the viewing frustum, it is possible to analytically compute the derivatives of the triangle's texture coordinates at that point given the camera matrix (including projection to the screen), the 3D coordinates of the triangle's vertices, the texture coordinates associated with each vertex, and the point on the triangle.[1] Our approach is based on first computing the derivatives of the barycentric coordinates at the point on the triangle and then computing the derivatives of the texture coordinates using the chain rule. (A related approach is described by Ewins et al. [4], who also evaluate a number of approximations such as derivatives that are constant over each triangle.)

[1] The technique can be generalized to shapes besides triangles, though we will focus on triangles here.

We will describe the method in the context of directly visible points with ray tracing, but it applies equally well to texture lookups during deferred shading. At the end of this chapter, we will discuss how this technique can be used to compute approximate texture filter widths for secondary intersections in a ray tracer.

5.2.1 INPUTS AND NOTATION

We expect that the world-space intersection point of the camera ray with the scene geometry and the camera matrix **M** is provided. We will also assume that the original triangle vertices X_0, X_1, and X_2 are available in homogeneous coordinates. Because we have the matrix **M** that is the combined projection from world space to screen shape, our strategy is to first evaluate the screen-space derivatives of the barycentric coordinates (u, v) of the intersection point. Once these are known, we can readily obtain the screen-space derivatives of the desired texture coordinates via the chain rule. (More generally, the barycentric derivatives can be used to compute the derivatives of any interpolated per-vertex value at the point.)

If not already available, both barycentric coordinates can be found by computing the areas of two triangles formed by two of the triangle's vertices and the ray intersection point and dividing them by the triangle's total area. These barycentric coordinates can equivalently be computed in world space or using the 2D screen-space projection of the corresponding points.

In the following, symbols with a tilde denote the screen-space projection of the corresponding variable:

$$\tilde{X} = \mathbf{M}X. \tag{5.1}$$

We will make frequent use of the symbol w to denote the linear form (row vector) $\mathbf{w} = (0, 0, 0, 1)$. Finally, when used, superscript indices will signify coordinates.

5.2.2 OVERVIEW

In order to obtain derivatives of texture coordinates, we use the relation between texture coordinates and barycentric coordinates. Given the three texture coordinates (s_i, t_i) at the vertices of a triangle, the (s, t) texture coordinates for the point corresponding to the barycentric coordinates (u, v) are

$$\begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} s_0 \\ t_0 \end{pmatrix} + \begin{pmatrix} s_1 - s_0 & s_2 - s_0 \\ t_1 - t_0 & t_2 - t_0 \end{pmatrix} \cdot \begin{pmatrix} u \\ v \end{pmatrix}. \tag{5.2}$$

We are interested in finding the derivatives of (s, t) with respect to (x, y) coordinates at a point $\tilde{X}$ in screen space, which is

$$\frac{\mathrm{d}\,(s,t)}{\mathrm{d}\left(\tilde{X}^1,\tilde{X}^2\right)} = \begin{pmatrix} s_1 - s_0 & s_2 - s_0 \\ t_1 - t_0 & t_2 - t_0 \end{pmatrix} \cdot \frac{\mathrm{d}\,(u,v)}{\mathrm{d}\left(\tilde{X}^1,\tilde{X}^2\right)}. \tag{5.3}$$

Given barycentric derivatives, the texture derivatives are easily computed using the following shader code:

```
float2x2 ComputeTexCoordDerivatives(
        float2x2 dst_dx1x2, float2 st0, float2 st1, float2 st2) {
    float2x2 dtc_duv = float2x2(st1.x - st0.x, st2.x - st0.x,
                                st1.y - st0.y, st2.y - st0.y);
    return mul(dtc_duv, dst_dx1x2);
}
```

Our task, then, is to compute the derivative on the right-hand side of Equation 5.3. Before we consider evaluating this expression, there is an important technical detail: the coordinates of $\tilde{X}$ are not independent, and so the derivative cannot be evaluated as a partial derivative. The dependency is in the third component (depth), which is determined completely by the first two components through linear screen-space interpolation in the triangle.

Switching to partial derivatives using the chain rule, we have

$$\frac{\mathrm{d}\,(u,v)}{\mathrm{d}\left(\tilde{X}^1,\tilde{X}^2\right)} = \frac{\partial\,(u,v)}{\partial\tilde{X}} \cdot \frac{\mathrm{d}\tilde{X}}{\mathrm{d}\left(\tilde{X}^1,\tilde{X}^2\right)}. \tag{5.4}$$

Applying the chain rule again to the first factor on the right-hand side gives

$$\frac{\mathrm{d}\,(u,v)}{\mathrm{d}\left(\tilde{X}^1,\tilde{X}^2\right)} = \frac{\partial\,(u,v)}{\partial X} \cdot \frac{\partial X}{\partial\tilde{X}} \cdot \frac{\mathrm{d}\tilde{X}}{\mathrm{d}\left(\tilde{X}^1,\tilde{X}^2\right)}. \tag{5.5}$$

Though there are some subtleties lurking under the notation, the three factors on the right-hand side of Equation 5.5 give the road map for our task. Reading left to right, first we will find the derivative of the barycentric coordinates in terms of the world-space point X. Next, we will convert those to be derivatives in terms of screen space $\tilde{X}$. Finally, we disentangle the relationship between (x, y) individually in screen space and the complete screen-space point $\tilde{X}$, giving independent derivatives in x and y.

We will consider each of these factors in turn.

5.2.3 WORLD-SPACE DERIVATIVES

For the first factor on the right-hand side of Equation 5.5, we start by defining $\mathbf{A}$ as the 2×3 matrix of two vectors corresponding to two edges of the triangle:

$$\mathbf{A} = \begin{bmatrix} X_1 - X_0 & X_2 - X_0 \end{bmatrix}. \tag{5.6}$$

In turn, the point X inside the triangle with barycentric coordinates u, v is given by

$$\mathbf{A} \begin{pmatrix} u \\ v \end{pmatrix} = X - X_0. \tag{5.7}$$

The partial derivative $\partial(u, v)/\partial X$ is therefore given by

$$\frac{\partial (u, v)}{\partial X} = \mathbf{A}^+, \tag{5.8}$$

where $\mathbf{A}^+$ is the Moore-Penrose inverse, also known as just the pseudoinverse matrix. Although $\mathbf{A}^+$ is 2×4, the relevant 2×3 part of it has a simple closed form. If we here denote the columns of $\mathbf{A}$ as A_1 and A_2, and the cross product is denoted by $\times$, then we have

$$\hat{N} = A_1 \times A_2, \tag{5.9}$$

$$\mathbf{A}^+ = \frac{1}{\hat{N} \cdot \hat{N}} \begin{bmatrix} A_2 \times \hat{N} \\ \hat{N} \times A_1 \end{bmatrix}. \tag{5.10}$$

The function that computes $\mathbf{A}^+$ and then returns the partial derivatives of the barycentric coordinates with respect to world-space coordinates follows directly.

```
1 void BarycentricWorldDerivatives(
2         float3 A1, float3 A2,
3         out float3 du_dx, out float3 dv_dx) {
4     float3 Nt = cross(A1, A2) / dot(Nt, Nt);
5     du_dx = cross(A2, Nt);
6     dv_dx = cross(Nt, A1);
7 }
```

5.2.4 FROM WORLD SPACE TO SCREEN SPACE

The next step is to find the middle factor on the right-hand side of Equation 5.5. The unprojected Cartesian point X can be found using the inverse of the camera matrix $\mathbf{M}$ and dividing by the inverse of the projected homogeneous component,

$$x = \frac{\mathbf{M}^{-1}\tilde{X}}{\mathbf{w}\mathbf{M}^{-1}\tilde{X}}, \tag{5.11}$$

and the partial derivative can be readily obtained from this equation using the product rule,

$$\frac{\partial X}{\partial \tilde{X}} = \frac{\mathbf{M}^{-1}}{\mathbf{w}\mathbf{M}^{-1}\tilde{X}} - \frac{\mathbf{M}^{-1}\tilde{X}\mathbf{w}\mathbf{M}^{-1}}{\left(\mathbf{w}\mathbf{M}^{-1}\tilde{X}\right)^2}, \tag{5.12}$$

which simplifies to

$$\frac{\partial X}{\partial \tilde{X}} = \left(\mathbf{w}\mathbf{M}X\right)\left(\mathbf{I} - X\mathbf{w}\right)\mathbf{M}^{-1}. \tag{5.13}$$

The corresponding implementation is straightforward:

```
float3x3 WorldScreenDerivatives(
        float4x4 WorldToTargetMatrix, float4x4 TargetToWorldMatrix,
        float4 x) {
    float wMx = dot(WorldToTargetMatrix[3], x);
    float3x3 dx_dxt = (float3x3)TargetToWorldMatrix;
    dx_dxt[0] -= x.x * TargetToWorldMatrix[3].xyz;
    dx_dxt[1] -= x.y * TargetToWorldMatrix[3].xyz;
    dx_dxt[2] -= x.z * TargetToWorldMatrix[3].xyz;
    return dx_dxt;
}
```

5.2.5 DEPTH DERIVATIVES

Finally, the derivatives of screen-space depth with respect to screen-space x and y must be found to calculate the last factor on the right-hand side of Equation 5.5. These derivatives can be evaluated explicitly for triangles, because in that case the depth is linear in screen space. However, a more general approach is taken to ensure applicability for other surfaces.

Locally the surface can be expressed in implicit form as

$$F\left(x\left(\tilde{X}\right)\right) = 0. \tag{5.14}$$

By the implicit function theorem,

$$\frac{\partial \tilde{X}^3}{\partial\left(\tilde{X}^1, \tilde{X}^2\right)} = -\left(\frac{\mathrm{d}F}{\mathrm{d}\tilde{X}^3}\right)^{-1} \cdot \frac{\mathrm{d}F}{\mathrm{d}\left(\tilde{X}^1, \tilde{X}^2\right)}, \tag{5.15}$$

whereas

$$\frac{\mathrm{d}F}{\mathrm{d}\tilde{X}} = \frac{\partial F}{\partial X} \cdot \frac{\partial X}{\partial \tilde{X}}. \tag{5.16}$$

Here x is a homogeneous vector, so the last coordinate doesn't appear explicitly in F; hence, there's no dependency on it in F. We can obtain

$$\frac{\partial F}{\partial X} = \gamma\left[\mathbf{n}, \quad 0\right], \tag{5.17}$$

where n is the linear form of the surface unit normal and γ is some scale of it. Combining Equations 5.13, 5.16, and 5.17, we get

$$\frac{\mathrm{d}F}{\mathrm{d}\tilde{X}} = (\mathbf{w}\mathbf{M}X)\,\gamma \begin{bmatrix} \mathbf{n}, & \begin{bmatrix} -\mathbf{n}, & 0 \end{bmatrix} \cdot X \end{bmatrix} \mathbf{M}^{-1}. \tag{5.18}$$

Then, we denote

$$\tilde{n} = \begin{bmatrix} \mathbf{n}, & \begin{bmatrix} -\mathbf{n}, & 0 \end{bmatrix} \cdot X \end{bmatrix} \mathbf{M}^{-1}. \tag{5.19}$$

Now recombining the components into Equation 5.15 gives

$$\frac{\partial \tilde{X}^3}{\partial \left(\tilde{X}^1, \tilde{X}^2\right)} = -\frac{\tilde{\mathbf{n}}_{1,2}}{\tilde{\mathbf{n}}_3}. \tag{5.20}$$

The subscripts are the covariant coordinates. Thus, the depth derivatives have been obtained if the normal of the surface at the point is known:

```
float2 DepthGradient(
        float4 x, float3 n, float4x4 TargetToWorldMatrix) {
    float4 n4 = float4(n, 0);
    n4.w = -dot(n4.xyz, x.xyz);
    n4 = mul(n4, TargetToWorldMatrix);
    n4.z = max(abs(n4.z), 0.0001) * sign(n4.z);
    return n4.xy / -n4.z;
}
```

5.2.6 PUTTING IT ALL TOGETHER

Combining Equations 5.8, 5.13, and 5.20 into Equation 5.5 yields the desired derivatives of barycentric coordinates with respect to screen-space coordinates. The following function uses the previously defined functions to compute the corresponding values and returns the derivatives:

```
float2x2 BarycentricDerivatives(
        float4 x, float3 n, float3 x0, float3 x1, float3 x2,
        float4x4 WorldToTargetMatrix, float4x4 TargetToWorldMatrix) {
    // Derivatives of barycentric coordinates with respect to
    // world-space coordinates (Section 5.2.3).
    float3 du_dx, dv_dx;
    BarycentricWorldDerivatives(x1 - x0, x2 - x0, du_dx, dv_dx);

    // Partial derivatives of world-space coordinates with respect
    // to screen-space coordinates (Section 5.2.4). (Only the
    // relevant 3x3 part is considered.)
    float3x3 dx_dxt = WorldScreenDerivatives(WorldToTargetMatrix,
                                             TargetToWorldMatrix, x);

    // Partial derivatives of barycentric coordinates with respect
    // to screen-space coordinates.
    float3 du_dxt = du_dx.x * dx_dxt[0] + du_dx.y * dx_dxt[1] +
                    du_dx.z * dx_dxt[2];
    float3 dv_dxt = dv_dx.x * dx_dxt[0] + dv_dx.y * dx_dxt[1] +
                    dv_dx.z * dx_dxt[2];
```

```
21
22      // Derivatives of barycentric coordinates with respect to
23      // screen-space x and y coordinates (Section 5.2.5).
24      float2 ddepth_dXY = DepthGradient(x, n, TargetToWorldMatrix);
25      float wMx = dot(WorldToTargetMatrix[3], x);
26      float2 du_dXY = (du_dxt.xy + du_dxt.z * ddepth_dXY) * wMx;
27      float2 dv_dXY = (dv_dxt.xy + dv_dxt.z * ddepth_dXY) * wMx;
28      return float2x2(du_dXY, dv_dXY);
29  }
```

5.3 FURTHER APPLICATIONS

5.3.1 TRILINEAR SAMPLING

If only trilinear sampling is needed, it is possible to obtain an expression for the fractional mip level directly. The matrix on the left-hand side in Equation 5.3 transforms tangents in screen space to tangents in texture space. Consider only unit length tangents v:

$$u = \frac{\mathrm{d}\left(w \cdot s, h \cdot t\right)}{\mathrm{d}\left(\tilde{X}^1, \tilde{X}^2\right)} \cdot v, \tag{5.21}$$

where w and h are the width and the height of the texture, respectively. A question that can be asked is: what is the resulting direction (of u) that yields the minimum and maximum scaling of the vector v? These two directions are the anisotropy directions for texturing sampling, whereas the scaling factors are the anisotropy magnitudes. These directions and magnitudes are the left singular vectors and the singular values (of the matrix), respectively. If only trilinear sampling is desired, then only the singular values are needed to determine the needed mip level to sample. At this point there are several options to choose from:

- Use the maximum singular value.
- Use the minimum singular value.
- Use any other value in the singular value spectrum.

Using any value that is not within the singular value spectrum will definitely lead to either blurring or aliasing. Using the maximum singular value will produce the smoothest, but possibly blurry, results. Using the minimum singular value will produce the sharpest, but also potentially aliased, results. It is also possible to interpolate between the two to find some middle ground.

The actual mip level is the base-two logarithm of the selected value.

5.3.2 SECONDARY RAY INTERSECTION POINTS

It is necessary to carry some information with the ray in order to compute accurate filter kernels at secondary intersection points, especially after scattering from curved specular surfaces. The approaches described earlier can be applied, or a more sophisticated method like Belcour et al.'s [3] can be used.

However, in many cases approximating the texture coordinate derivatives at secondary points using the same derivatives as would have if they were directly visible works reasonably well [6]. Our technique requires that the visible point be inside the viewing frustum, however, which is not necessarily the case for secondary intersection points. In that case, the point can be rotated so that it lies inside the viewing frustum; placing it along the viewing axis works well. Möller and Hughes [7] describe an efficient algorithm for generating such rotation matrices.

5.3.3 MATERIAL GRAPHS

When materials are evaluated through a material graph (as opposed to a fixed material), then the derivatives can be used as inputs to the graph system. If the system supports automatic differentiation, then the derivatives for every sample can be evaluated correctly.

5.4 COMPARISON

Figures 5-1 and 5-2 illustrate the effect of anisotropic filtering versus bilinear filtering from mip level 0. Figure 5-3 shows accumulated bilinear filtering and serves as a reference.

5.5 CONCLUSION

We have presented an efficient method for computing texture derivatives for ray tracing and deferred shading that is based purely on the local geometry of a visible point and the camera's projection matrix. It does not directly handle secondary ray intersections and will give inaccurate filter width estimates in that case, especially in the presence of specular reflections. However, for directly visible points and for applications where that is not a concern, our approach is effective.

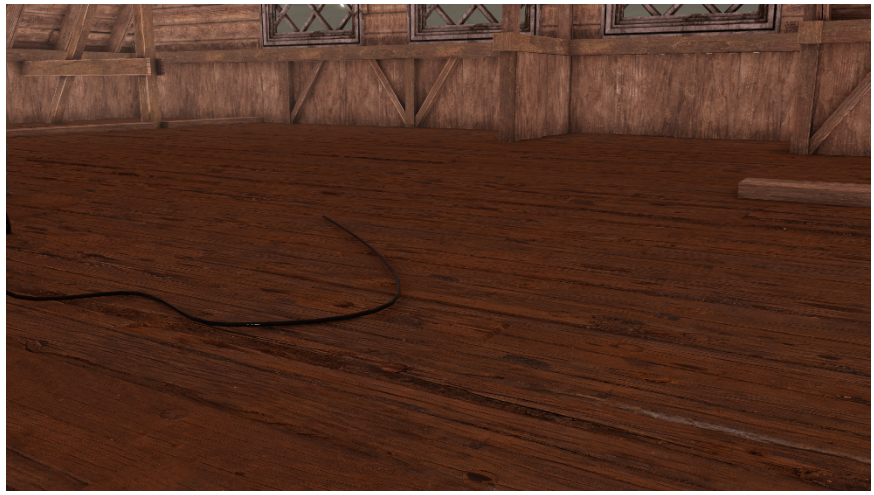


Figure 5-1. *Anisotropic filtering.*

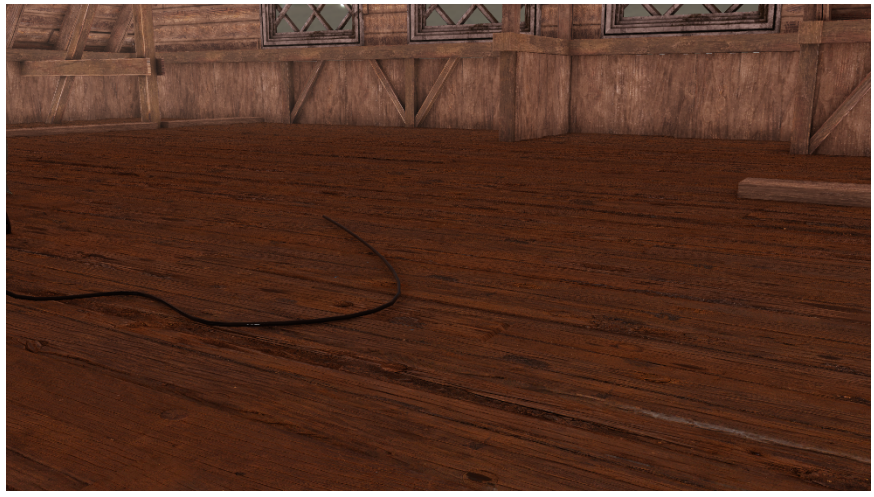


Figure 5-2. *Bilinear filtering from mip level 0.*

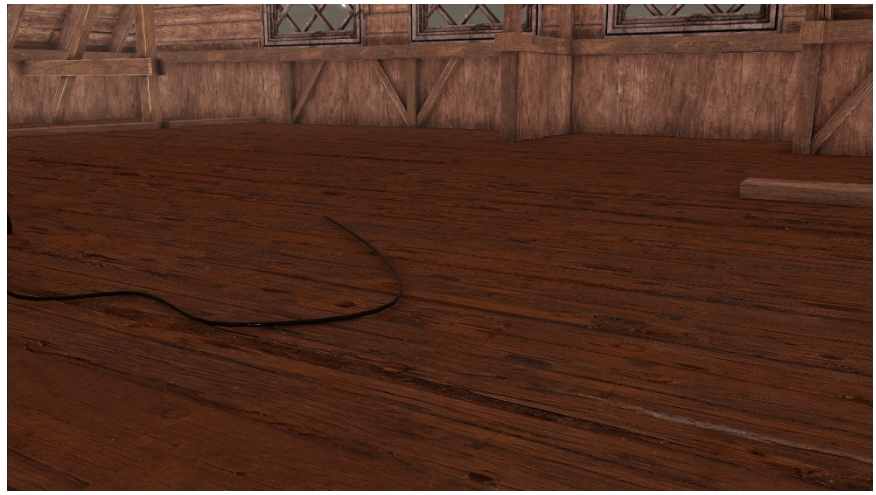


Figure 5-3. *Accumulated bilinear filtering from mip level 0.*

REFERENCES

[1] Akenine-Möller, T., Crassin, C., Boksansky, J., Belcour, L., Panteleev, A., and Wright, O. Improved shader and texture level of detail using ray cones. *Journal of Computer Graphics Techniques (JCGT)*, 10(1):1–24, 2021. http://jcgt.org/published/0010/01/01/.

[2] Akenine-Möller, T., Nilsson, J., Andersson, M., Barré-Brisebois, C., Toth, R., and Karras, T. Texture level of detail strategies for real-time ray tracing. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 321–345. Apress, 2019. DOI: 10.1007/978-1-4842-4427-2_20.

[3] Belcour, L., Yan, L.-Q., Ramamoorthi, R., and Nowrouzezahrai, D. Antialiasing complex global illumination effects in path-space. *ACM Transactions on Graphics*, 36(4), Jan. 2017. DOI: 10.1145/3072959.2990495.

[4] Ewins, J. P., Waller, M. D., White, M., and Lister, P. F. Mip-map level selection for texture mapping. *IEEE Transactions on Visualization and Computer Graphics*, 4(4):317–329, Oct. 1998. DOI: 10.1109/2945.765326.

[5] Igehy, H. Tracing ray differentials. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 179–186, 1999. DOI: 10.1145/311535.311555.

[6] Li, Y. K. Mipmapping with bidirectional techniques. https://blog.yiningkarlli.com/2018/10/bidirectional-mipmap.html, October 28, 2018.

[7] Möller, T. and Hughes, J. F. Efficiently building a matrix to rotate one vector to another. *Journal of Graphics Tools*, 4(4):1–4, 1999. DOI: 10.1080/10867651.1999.10487509.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 6

DIFFERENTIAL BARYCENTRIC COORDINATES

Tomas Akenine-Möller
NVIDIA

ABSTRACT

We describe an optimized way to compute differential barycentric coordinates, which can be employed by implementations of ray differentials. This technique can be used for texture filtering computations for ray tracing.

6.1 BACKGROUND

A ray is described as $R(t) = O + t\mathbf{d}$, where O is the ray origin, $\mathbf{d}$ is the normalized ray direction, and t is the distance along the ray from the origin. A ray differential [3] is useful for tracking the size of the footprint along a ray path and is described by $(\frac{\partial O}{\partial x}, \frac{\partial O}{\partial y}, \frac{\partial \mathbf{d}}{\partial x}, \frac{\partial \mathbf{d}}{\partial y})$, where x and y are pixel coordinates. This is illustrated in Figure 6-1.

Computing differential barycentric coordinates is an important part of any ray differential implementation. To be able to compute the differential barycentric coordinates at a hit point defined by t, we need to compute the differential origin, $(\frac{\partial O'}{\partial x}, \frac{\partial O'}{\partial y})$, at the hit point. We first describe how this is done and then use that result in Section 6.2 to compute differential barycentric coordinates.

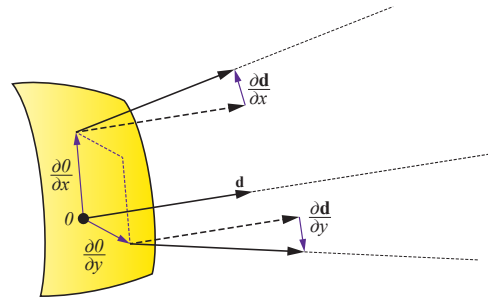


Figure 6-1. *A ray is defined by an origin, O, and a direction,* $\mathbf{d}$*. A ray differential consists of differentials of both the ray origin and the ray direction, i.e.,* $(\frac{\partial O}{\partial x}, \frac{\partial O}{\partial y}, \frac{\partial \mathbf{d}}{\partial x}, \frac{\partial \mathbf{d}}{\partial y})$*. These together form a beam, which is traced through the scene.*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_6

Assuming that the current ray differential is $(\frac{\partial O}{\partial x}, \frac{\partial O}{\partial y}, \frac{\partial \mathbf{d}}{\partial x}, \frac{\partial \mathbf{d}}{\partial y})$, we need to propagate the ray differential along the ray, $R(t) = O + t\mathbf{d}$, to the first hit point, at distance t from the origin, O. The normalized interpolated normal at the hit point is called $\mathbf{n}_i$. Propagation of the differential ray origin [3] is done as

$$\frac{\partial O'}{\partial x} = \frac{\partial T}{\partial x} - \frac{\frac{\partial T}{\partial x} \cdot \mathbf{n}_i}{\mathbf{d} \cdot \mathbf{n}_i}\mathbf{d}, \tag{6.1}$$

where $\frac{\partial T}{\partial x} = \frac{\partial O}{\partial x} + t\frac{\partial \mathbf{d}}{\partial x}$ is the differential width in x of the ray beam at the hit point, without projection onto the normal at the hit point. A similar expression is used for $\frac{\partial T}{\partial y}$. Note that $\frac{\partial \mathbf{d}}{\partial x}$ and $\frac{\partial \mathbf{d}}{\partial y}$ are not updated during propagation. Also, for eye rays, we have $\frac{\partial O}{\partial x/y} = (0, 0, 0)$, which means that Equation 6.1 becomes a little simpler in the first step.

Now, it only remains to compute the differential ray direction, $(\frac{\partial \mathbf{d}}{\partial x}, \frac{\partial \mathbf{d}}{\partial y})$, of the eye ray, which is needed for $\frac{\partial T}{\partial x}$. (This part can also be found in a chapter in *Ray Tracing Gems* [1].) In the DirectX Raytracing API, a common way to compute a normalized camera ray direction is

$$\mathbf{p} = (p_x, p_y) = \left(2 \cdot \frac{x + 0.5}{w} - 1, -2 \cdot \frac{y + 0.5}{h} + 1\right), \tag{6.2}$$

$$\mathbf{g} = p_x\mathbf{r} + p_y\mathbf{u} + \mathbf{v}, \tag{6.3}$$

$$\mathbf{d} = \frac{\mathbf{g}}{\|\mathbf{g}\|}, \tag{6.4}$$

where $w \times h$ is the resolution of the image, pixel $(0, 0)$ is at the upper left corner of the image, and (x, y) are the integer pixel coordinates. Note that $(x + 0.5)/w$ is in $[0, 1]$ and that 0.5 has been added to get to the pixel center, so we get a perfect match between rasterization and ray tracing. These 0.5 terms can be replaced with random values in $[0, 1]$ for jittering, if desired. The final values for $\mathbf{p}$ are in $[-1, 1]$. The camera orientation is given by the vectors $(\mathbf{r}, \mathbf{u}, \mathbf{v})$, which are right, up, and view, respectively. In our case, these are $(\mathbf{r}, \mathbf{u}, \mathbf{v}) = (af\mathbf{r}', f\mathbf{u}', \mathbf{v}')$, where a is the aspect ratio, $f = \tan(\omega/2)$, where ω is the vertical field of view, and $(\mathbf{r}', \mathbf{u}', \mathbf{v}')$ are the normalized camera frame vectors. In Falcor [2], which is the research platform where we have implemented our methods, $(\mathbf{r}, \mathbf{u}, \mathbf{v})$ are all multiplied by the focal distance, but since we normalize $\mathbf{d}$ in Equation 6.4, we have omitted that factor here. It is important to specify how $\mathbf{d}$ is computed because its differential depends on it. Differentiating Equation 6.4, we get

$$\frac{\partial \mathbf{d}}{\partial x} = \frac{2(k_g\mathbf{r} - k_r\mathbf{g})}{wk_g^{3/2}}, \quad \frac{\partial \mathbf{d}}{\partial y} = \frac{-2(k_g\mathbf{u} - k_u\mathbf{g})}{hk_g^{3/2}}, \tag{6.5}$$

where $k_g = \mathbf{g} \cdot \mathbf{g}$, $k_r = \mathbf{g} \cdot \mathbf{r}$, $k_u = \mathbf{g} \cdot \mathbf{u}$, and $\mathbf{g}$ is specified by Equation 6.3.

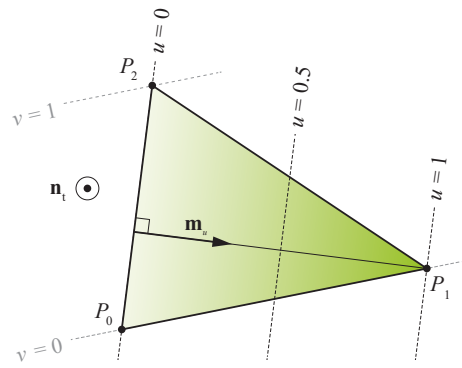


Figure 6-2. *A triangle with three vertices P_0, P_1, and P_2. The barycentric coordinates (u, v) are also visualized, together with the "normal" $\mathbf{m}_u$ of the edge through P_0 and P_2. The triangle plane normal, $\mathbf{n}_t$, points outward toward the reader, orthogonal to the paper/display.*

6.2 METHOD

Next, we describe how to compute the differentials of the barycentric coordinates, (u, v), at a hit point inside a triangle.

The vertices of the triangle are P_0, P_1, and P_2, and we assume that the triangle normal is $\mathbf{n}_t$. To compute differential barycentrics, Igehy [3, page 182] teaches that edge equations that go through the edges should be computed, and those should evaluate to 0.0 for points on the edge and to 1.0 for a vertex not being an edge vertex. Exactly how these edge equations are constructed was left out. We describe the details of one way of doing this and some optimizations. Let us start with the edge equation going through P_0 and P_2, which is connected to the barycentric coordinate u, as shown in Figure 6-2. A reasonable approach is to compute a "normal," here called $\mathbf{m}_u$, of the edge that is also perpendicular to the triangle normal $\mathbf{n}_t$, i.e.,

$$\mathbf{m}_u = (P_2 - P_0) \times \mathbf{n}_t = \mathbf{e}_2 \times \mathbf{n}_t. \tag{6.6}$$

Similarly for v, we have $\mathbf{m}_v = (P_1 - P_0) \times \mathbf{n}_t = \mathbf{e}_1 \times \mathbf{n}_t$. The plane equation is then $\mathbf{m}_u \cdot (X - P_0) = 0$, where X is any point on the plane with normal $\mathbf{m}_u$. The edge equation then becomes $e_u(X) = \mathbf{m}_u \cdot (X - P_0)$. However, it needs to be normalized so $e_u(P_1) = 1$, which can be obtained as $e'_u(X) = e_u(X)/e_u(P_1)$. As it turns out, only the normal vector of the plane equation is needed for these computations [3], and this normalized normal, here called $\mathbf{l}_u$, is then

$$\mathbf{l}_u = \frac{1}{\mathbf{m}_u \cdot (P_1 - P_0)}\mathbf{m}_u = \frac{1}{\mathbf{m}_u \cdot \mathbf{e}_1}\mathbf{m}_u, \tag{6.7}$$

and similarly for v, we get

$$\mathbf{l}_v = \frac{1}{\mathbf{m}_v \cdot (P_2 - P_0)}\mathbf{m}_v = \frac{1}{\mathbf{m}_v \cdot \mathbf{e}_2}\mathbf{m}_v. \tag{6.8}$$

The triangle plane normal $\mathbf{n}_t$ can be computed as $\mathbf{n}_t = \mathbf{e}_1 \times \mathbf{e}_2$, which does not need to be normalized for our purposes because we already normalized the plane equations in Equations 6.7 and 6.8.

Igehy also computes a plane equation for the third barycentric coordinate w, but since $w = 1 - u - v$, we have $\partial w/\partial x = -\partial u/\partial x - \partial v/\partial x$, which we choose to exploit because it is faster. The differential barycentric coordinates are then [3]

$$\left(\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial v}{\partial y}\right) = \left(\mathbf{l}_u \cdot \frac{\partial O'}{\partial x}, \mathbf{l}_v \cdot \frac{\partial O'}{\partial x}, \mathbf{l}_u \cdot \frac{\partial O'}{\partial y}, \mathbf{l}_v \cdot \frac{\partial O'}{\partial y}\right), \tag{6.9}$$

where we use Equation 6.1 for the differential of the ray origin, $(\frac{\partial O'}{\partial x}, \frac{\partial O'}{\partial y})$.

Finally, let us assume that the texture coordinates at the vertices are denoted $\mathbf{t}_i = (t_{ix}, t_{iy})$. To be able to use anisotropic texture filtering available on GPUs, we need the differentials of the texture coordinates. Recall that the texture coordinates are interpolated as $\mathbf{t}(u, v) = (1 - u - v)\mathbf{t}_0 + u\mathbf{t}_1 + v\mathbf{t}_2$. This expression can be differentiated, and we can use the differential barycentric coordinates (Equation 6.9) to find the differential texture coordinates as

$$\frac{\partial \mathbf{t}}{\partial x} = \left(-\frac{\partial u}{\partial x} - \frac{\partial v}{\partial x}\right)\mathbf{t}_0 + \frac{\partial u}{\partial x}\mathbf{t}_1 + \frac{\partial v}{\partial x}\mathbf{t}_2 = \frac{\partial u}{\partial x}(\mathbf{t}_1 - \mathbf{t}_0) + \frac{\partial v}{\partial x}(\mathbf{t}_2 - \mathbf{t}_0), \tag{6.10}$$

and similarly for $\frac{\partial \mathbf{t}}{\partial y}$. We provide an implementation of ray differentials in Falcor [2]. The important parts can be found in the `TexLODHelpers.slang` file, with some functions listed in the next section.

6.3 CODE

The `RayDiff` struct together with the most important functions are listed here. The key function is `computeBarycentricDifferentials()`, which implements Equations 6.7–6.9. Its results are stored in the output variables `dBarydx` and `dBarydy`. The input to `computeBarycentricDifferentials()` deserves some attention: `edge01` and `edge02` are triangle edges in world space, which can be computed as follows:

```
edge01 = mul(vertices[1].pos - vertices[0].pos, (float3x3)worldMat);
edge02 = mul(vertices[2].pos - vertices[0].pos, (float3x3)worldMat);
triNormalW = cross(edge01, edge02);
```

```
struct RayDiff
{
    float3 dOdx;
    float3 dOdy;
    float3 dDdx;
    float3 dDdy;

    static RayDiff create(float3 dOdx, float3 dOdy,
        float3 dDdx, float3 dDdy)
    {
        RayDiff rd;
        rd.dOdx = dOdx;   rd.dOdy = dOdy;
        rd.dDdx = dDdx;   rd.dDdy = dDdy;
        return rd;
    }

    // Implements Equation 6.1 of this chapter.
    RayDiff propagate(float3 O, float3 D, float t, float3 N)
    {
        float3 dodx = dOdx + t * dDdx;     // Igehy Equation 10
        float3 dody = dOdy + t * dDdy;
        float rcpDN = 1.0f / dot(D, N);    // Igehy Eqns 10 & 12
        float dtdx = -dot(dodx, N) * rcpDN;
        float dtdy = -dot(dody, N) * rcpDN;
        dodx += D * dtdx;
        dody += D * dtdy;

        return RayDiff.create(dodx, dody, dDdx, dDdy);
    }
};

// Implements Equation 6.10 of this chapter.
void interpolateDifferentials(float2 dBarydx, float2 dBarydy,
    float2 vertexValues[3], out float2 dx, out float2 dy)
{
    float2 delta1 = vertexValues[1] - vertexValues[0];
    float2 delta2 = vertexValues[2] - vertexValues[0];
    dx = dBarydx.x * delta1 + dBarydx.y * delta2;
    dy = dBarydy.x * delta1 + dBarydy.y * delta2;
}

// Implements Equation 6.9 of this chapter.
void computeBarycentricDifferentials(RayDiff rayDiff,
    float3 rayDir, float3 edge01, float3 edge02,
    float3 triNormalW, out float2 dBarydx, out float2 dBarydy)
{
    float3 Nu = cross(edge02, triNormalW);
    float3 Nv = cross(edge01, triNormalW);
    float3 Lu = Nu / (dot(Nu, edge01));
    float3 Lv = Nv / (dot(Nv, edge02));

    dBarydx.x = dot(Lu, rayDiff.dOdx);     // du / dx
    dBarydx.y = dot(Lv, rayDiff.dOdx);     // dv / dx
    dBarydy.x = dot(Lu, rayDiff.dOdy);     // du / dy
    dBarydy.y = dot(Lv, rayDiff.dOdy);     // dv / dy
}
```

REFERENCES

[1] Akenine-Möller, T., Nilsson, J., Andersson, M., Barré-Brisebois, C., Toth, R., and Karras, T. Texture level-of-detail strategies for real-time ray tracing. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 20, pages 321–345. NVIDIA/Apress, 2019.

[2] Benty, N., Yao, K.-H., Foley, T., Kaplanyan, A. S., Lavelle, C., Wyman, C., and Vijay, A. The Falcor real-time rendering framework. https://github.com/NVIDIAGameWorks/Falcor, 2017. Accessed October 25, 2018.

[3] Igehy, H. Tracing ray differentials. In *Proceedings of SIGGRAPH 99*, pages 179–186, 1999.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 7

TEXTURE COORDINATE GRADIENTS ESTIMATION FOR RAY CONES

Wessam Bahnassi
NVIDIA

ABSTRACT

Ray cones were presented as a fast, approximate way to calculate a suitable mip level index for a particular texture sample in a ray tracing shader setup. Though the original technique was doing all the mathematics needed to compute the final mip level index, in this chapter we reformulate the problem in a different way that relies on existing hardware-supported instructions. This maintains almost the same visual results and leads to more convenient shader code. It also further simplifies and reduces the number of instructions from the original ray cones implementation, though we do not claim this necessarily leads to better performance.

7.1 BACKGROUND

Texture sampling under real-time ray tracing suffers the same limitations as when sampling textures in shader stages outside the pixel shader stage. The lack of texture coordinate derivatives inhibits the ability to rely on the hardware logic for choosing a suitable mip level index for the given sample. In particular, it is the hardware's inability to internally derive the screen-space derivatives that prevents using sampling functions like HLSL's `Texture.Sample()`, which internally calculates a suitable mip level index.

Techniques to manually compute the proper texture mip level index exist, the latest of which was published in *Ray Tracing Gems* [2]. In that chapter, two techniques were mentioned: ray differentials and ray cones. Ray cones are faster to compute than ray differentials, and they provide good-quality results for real-time rendering purposes.

This chapter details the process of computing a cone that follows the launched ray as it hits surfaces and uses the cone size at each hit location to compute a suitable mip level index for textures sampled for the hit triangle. The computation starts with a triangle-based level of detail (LOD) value and

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_7

adds to it two factors: distance-based and normal-based. The result of the full computation produces a mip level index that can be fed to functions like HLSL's `Texture.SampleLevel()`.

Because that method calculates a mip level index, the calculation—or at least part of it—is texture-specific. This makes sense as different texture dimensions will result in different mip level index values under the same sampling location. For ray cones, we can split the mathematics into two parts: UV-coordinates-dependent and texture-size-dependent. This way, it is possible to reduce redundant calculations when sampling multiple textures at the same UV coordinates. However, the texture-size-dependent part of the calculations must be repeated for each texture. This involves first finding out the texture dimensions in some way, either by passing them as shader constants or by calling functions like HLSL's `Texture.GetDimensions()`.

The following code listing shows a full HLSL implementation of the technique and illustrates pseudocode to sample a texture using the provided functions.

```
float2 TriUVInfoFromRayCone(
    float3 vRayDir, float3 vWorldNormal, float vRayConeWidth,
    float2 aUV[3], float3 aPos[3], float3x3 matWorld)
{
    float2 vUV10 = aUV[1]-aUV[0];
    float2 vUV20 = aUV[2]-aUV[0];
    float fTriUVArea = abs(vUV10.x*vUV20.y - vUV20.x*vUV10.y);

    float3 vEdge10 = mul(aPos[1]-aPos[0],matWorld);
    float3 vEdge20 = mul(aPos[2]-aPos[0],matWorld);
    float3 vFaceNrm = cross(vEdge10, vEdge20); // In world space

    // Triangle-wide LOD offset value
    float fTriLODOffset = 0.5f*log2(fTriUVArea/length(vFaceNrm));
    float fDistTerm = vRayConeWidth * vRayConeWidth;
    float fNormalTerm = dot(vRayDir, vWorldNormal);

    return float2(fTriLODOffset, fDistTerm/(fNormalTerm*fNormalTerm));
}

float TriUVInfoToTexLOD(uint2 vTexSize, float2 vUVInfo)
{
    return vUVInfo.x + 0.5f*log2(vTexSize.x * vTexSize.y * vUVInfo.y);
}

// Compute ray cone parameters for a particular UV coordinate set.
float2 vTriUVInfo = TriUVInfoFromRayCone(...);

// Then for each texture to be sampled on that
// UV coordinate set, do:
uint2 vSize;
tex.GetDimensions(vSize.x, vSize.y);
float fMipIndex = TriUVInfoToTexLOD(vSize, vTriUVInfo);
float4 vSample = tex.SampleLevel(sampler, vUV, fMipIndex);
```

In the next section we show that it is possible to reduce the calculations in the ray cones technique further.

7.2 RAY CONE GRADIENTS

Hardware support for texture sampling is offered at different levels. In HLSL terms, `Texture.SampleLevel()` represents the lowest-level sampling function available, where the caller is responsible for providing the mip level index, and the function can provide at most trilinear filtering (combining eight texels). On the high-level end, `Texture.Sample()` only needs a sampling location and relies on the hardware's internal computations for the mip level index, and can internally combine even more than eight texels if, for example, the specified sampler requested anisotropic sampling. As mentioned before, `Texture.Sample()` requires operating in conditions not satisfied in ray tracing shaders. In between the two extremes, HLSL provides `Texture.SampleGrad()`, which instead of taking a particular mip level index, takes the derivative information that `Texture.Sample()` internally requires to operate with all its capabilities. `Texture.SampleGrad()` is not subject to the restrictions that `Texture.Sample()` has. Thus, it can work just as well as `Texture.SampleLevel()` in a ray tracing shader. For ray cones, `Texture.SampleGrad()` has a few benefits over `Texture.SampleLevel()`:

1. Gradients are a property of the UV coordinate set only, whereas the mip level index is a product of both the UV coordinate set and the texture dimensions. Using `Texture.SampleGrad()` allows the shader to skip all the texture-dependent calculations (e.g., texture size and mip level index calculation). This means less shader instructions to execute and more convenient use as well.

2. Also, `Texture.SampleGrad()` can respect anisotropic filtering if the sampler demands it. This assumes that the provided derivatives have anisotropy in them. However, this chapter will not cover the anisotropic derivatives mathematics because the goal here is to provide fast calculations for the standard trilinear filtering case. The anisotropic sampling derivatives mathematics is unnecessary if only regular trilinear filtering is required. Please refer to Möller et al. [1] for details on supporting anisotropy for ray cones.

One possible disadvantage of `Texture.SampleGrad()` is that it is a considerably more complex instruction than `Texture.SampleLevel()`, and thus can be slower to execute.

In order to use `Texture.SampleGrad()`, the mathematics for ray cones must be modified to compute UV derivatives instead of a mip level index.

The concept is simple: The ray cone tracks the pixel size throughout the ray tracing process. At any hit point, the ray cone can be projected on the hit surface. The projected size represents how much triangle area the projected pixel covers, and thus how much UV coordinate area is covered by the pixel. Since the UV gradients are to be used for isotropic trilinear filtering, there is no need to be precise about the gradients' directions nor the angle between them. It is safe to assume that it is just 90°. In this case, the covered UV coordinate area can be assumed to be a square, where the square root of the area value will give the length of its side. The resulting gradients are thus any two perpendicular 2D vectors with the side length just calculated. For simplicity, we use (`side_length`, 0) and (0, `side_length`).

The following code listing shows an example implementation of this idea. It is immediately evident how the gradient calculations lead to fewer instructions. There is no need to interrogate the sampled texture for its dimensions anymore, and calls to `log2()` are removed because these were needed for mip level index calculation. Note that the calculations in this code listing assume a quad face instead of a triangle face. This saves a couple of unnecessary multiplies and does not affect the final result as the calculation looks for a coverage ratio only.

```
float4 UVDerivsFromRayCone(
    float3 vRayDir, float3 vWorldNormal, float vRayConeWidth,
    float2 aUV[3], float3 aPos[3], float3x3 matWorld)
{
    float2 vUV10 = aUV[1]-aUV[0];
    float2 vUV20 = aUV[2]-aUV[0];
    float fQuadUVArea = abs(vUV10.x*vUV20.y - vUV20.x*vUV10.y);

    float3 vEdge10 = mul(aPos[1]-aPos[0],matWorld);
    float3 vEdge20 = mul(aPos[2]-aPos[0],matWorld);
    float3 vFaceNrm = cross(vEdge10, vEdge20);
    float fQuadArea = length(vFaceNrm);

    float fNormalTerm = abs(dot(vRayDir,vWorldNormal));
    float fPrjConeWidth = vRayConeWidth/fNormalTerm;
    float fVisibleAreaRatio = (fPrjConeWidth*fPrjConeWidth)/fQuadArea;

    float fVisibleUVArea = fQuadUVArea*fVisibleAreaRatio;
    float fULength = sqrt(fVisibleUVArea);
    return float4(fULength,0,0,fULength);
}

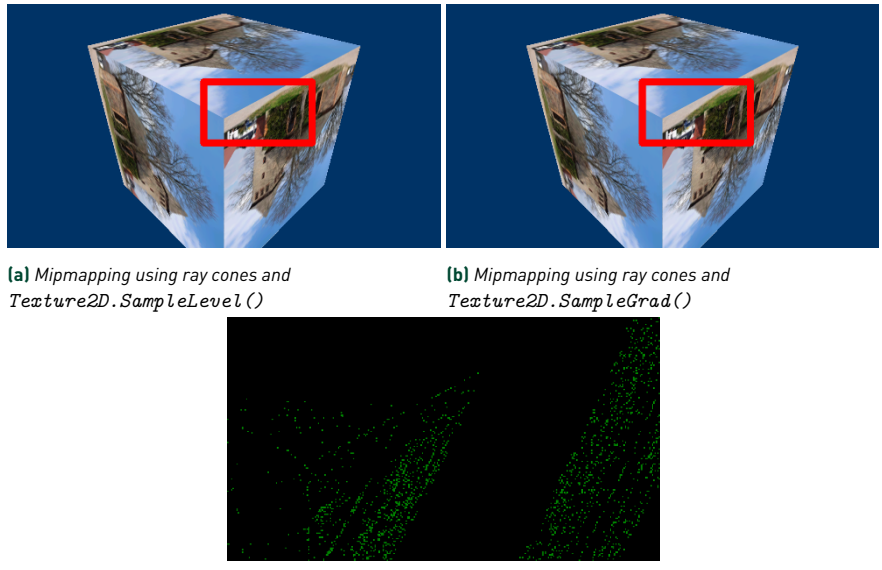
// Compute ray cone parameters for a particular UV coordinate set.
float4 vUVGrads = UVDerivsFromRayCone(...);
```

```
25
26 // Then for each texture to be sampled on that
27 // UV coordinate set, do:
28 float4 vSample = tex.SampleGrad(sampler, vUV, vUVGrads.xy, vUVGrads.zw);
```

7.3 COMPARISON AND RESULTS

Both `Texture2D.SampleLevel()` and `Texture2D.SampleGrad()` ray cone implementations were compared for quality and performance. The visual comparison shown in Figure7-1 demonstrates very minor differences between the two due to the mathematical reformulation for `Texture2D.SampleGrad()`. Please note that the original range in the difference image is between 0 and 1/255. The colors were scaled up greatly to make the differences distinguishable by human vision. The difference image zooms into the area marked by the red rectangle from the comparison images.

When both images are compared to a reference `Texture2D.Sample()` implementation, the difference ratio is similar for both ray cone implementations. There are two reasons for those differences. First, ray cones themselves use approximations to speed up the calculations, which means that both the original implementation and the gradients-based approach shown here will be subject to differences from a reference `Texture2D.Sample()`. Second, the hardware implementation for computing



(a) *Mipmapping using ray cones and `Texture2D.SampleLevel()`*

(b) *Mipmapping using ray cones and `Texture2D.SampleGrad()`*

(c) *Zoomed-in highlighted difference between the images (a) and (b)*

Figure 7-1. *Comparison images produced by the sample application.*

	Time	Mip 2	Mip 3	Mip 4	Mip 5
`Texture2D.Sample()`	7.15 ms	69,136	254,604	77,100	0
Original ray cones	21.23 ms	5,208	169,262	209,900	16,470
Ray cones with gradients	24.33 ms	5,202	169,234	209,924	16,480

Table 7-1. *Performance results collected on an NVIDIA RTX 3080 graphics card.*

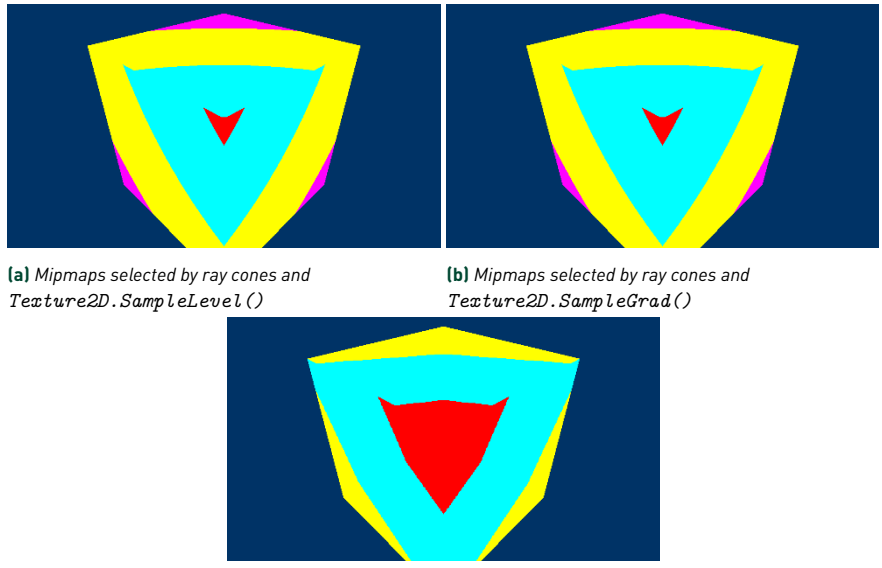
the mip level index in `Texture2D.Sample()` and `Texture2D.SampleGrad()` is most probably different than an implementation written manually in shader code, and can also differ between hardware vendors.

This slightly reduces the value of relying on `Texture2D.Sample()` as an absolute reference point for mipmapping results. For example, one approach might result in slight bias toward higher-detailed mipmaps, whereas another approach might bias to the opposite. Those differences can also affect performance depending on data access patterns and cache mechanisms.

The comparison data in Table 7-1 was gathered from drawing a simple static textured 3D cube 1000 times using instancing in one draw call. All instances are the same and drawn on top of each other with equal-greater depth testing to maximize the number of pixel shader invocations. This is important because the performance differences are otherwise too tiny to measure and can be easily lost in the noise in typical rendering scenes.

The pixel shader samples six different 2048 $\times$ 2048 32-bit (ARGB8888) textures using a bilinear sampler with point filtering for mipmaps. The textures have unique colors in each mip to allow counting of the number of pixels sampled from each mip. The table compares three ways of sampling the textures: the standard `Texture2D.Sample()` (which is possible in this test because it is run as a pixel shader), the original ray cone calculations and `Texture2D.SampleLevel()` (via the use of a simulated primary ray), and sampling using ray cones with gradients (`Texture2D.SampleGrad()`).

The time is the full scene rendering time measured on an RTX3080 GPU. The "Mip" columns in the table indicate the number of pixels produced from each mip level of the textures. The sum of the pixels on each row is the same for other rows. However, the distribution of pixels among mips differs depending on the technique used. Please note that each pixel samples the texture six times, multiplied by 1000 runs, so the actual number of texture samples in this case is 6000 multiplied by the numbers in the table.



(a) *Mipmaps selected by ray cones and* `Texture2D.SampleLevel()`

(b) *Mipmaps selected by ray cones and* `Texture2D.SampleGrad()`

(c) *Mipmaps selected by* `Texture2D.Sample()`

Figure 7-2. *Images produced by the performance test. The sampled texture's dimensions are* 2048 × 2048. *Red is mip 2, cyan is mip 3, yellow is mip 4, and purple is mip 5.*

Table 7-1 and Figure 7-2 show the differences between the three techniques. Obviously there is a big cost for ray cones compared to a straight `Texture2D.Sample()` instruction. However, it is interesting to see that the gradients approach incurs a performance hit compared to the original ray cone approach despite that, in general, the gradients approach seems to slightly favor less-detailed mipmaps in this particular test setup. When validating the produced shader disassembly, the gradients approach indeed generated half the number of instructions of the original ray cones for sampling the texture. This directs the performance difference toward the cost of the `Texture2D.SampleGrad()` instruction itself on the test GPU. The test scenario also fails to highlight the benefits of savings in arithmetic instructions because it is completely bound by texture sampling.

Keep in mind the heavy exaggeration performed by the test. Typical scenes do not usually incur 1000 layers of overdraw. In typical scenes the performance difference between the original ray cones and gradients might not be evident at all, but the convenience of the simplified code and sampling with `Texture2D.SampleGrad()` is a nice gain by itself.

7.4 SAMPLE CODE

The sample code for the techniques outlined in this chapter is available on the book's source code GitHub repository. The application is written for D3D12. It has multiple code paths to showcase the differences between all the techniques referenced here. The scene is made of a single cube viewed from an orbiting camera and is drawn using either rasterization or ray tracing, depending on the mode selected by the user. The modes are as follows:

1. *Mode 1:* Draw using rasterization and use the standard `Texture2D.Sample()` HLSL instruction. This is the reference for mipmapping quality. This is also the application's initial mode.
2. *Mode 2:* Draw using ray tracing and sample the texture using `Texture2D.SampleLevel()`, where the mip level index is fixed to level 0 (the most detailed level). The visual results suffer from heavy aliasing due to lack of mipmapping.
3. *Mode 3:* Draw using ray tracing and sample the texture using `Texture2D.SampleLevel()`, where the mip level index is computed according to the original ray cone implementation.
4. *Mode 4:* Draw using ray tracing and sample the texture using `Texture2D.SampleGrad()`, where the mip level index is computed following the new approach described in this chapter.

To toggle between the modes, simply press the mode's corresponding number on the keyboard (1–4).

7.5 CONCLUSION

Ray cones are a simple and fast way to support mipmapped texturing under environments where screen derivatives are not available, such as ray tracing shaders. In this chapter we presented a ray cone implementation that relies on `Texture2D.SampleGrad()` as a more convenient alternative to the split approach needed by the original ray cones and `Texture2D.SampleLevel()`. The visual results are very similar to the original ray cone results. This approach reduces the arithmetic from the original ray cone implementation, but the use of the `Texture2D.SampleGrad()` instruction has been shown to have a performance cost that can outweigh the arithmetic instructions savings depending on the application's bottleneck.

REFERENCES

[1] Akenine-Möller, T., Crassin, C., Boksansky, J., Belcour, L., Panteleev, A., and Wright, O. Improved shader and texture level of detail using ray cones. *Journal of Computer Graphics Techniques*, 10(1):1–24, 2021. http://jcgt.org/published/0010/01/01/.

[2] Akenine-Möller, T., Nilsson, J., Andersson, M., Barré-Brisebois, C., Toth, R., and Karras, T. Texture level of detail strategies for real-time ray tracing. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 20, pages 321–345. Apress, 2019.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 8

REFLECTION AND REFRACTION FORMULAS

Eric Haines
NVIDIA

ABSTRACT

Surfaces reflect. Glass, water, and similar materials both reflect and refract. The reflection and refraction formulas are at the heart of ray tracing. These are not easy to find in the literature in the vector forms needed for generating rays. This chapter is a reference for determining outgoing ray directions under simplified, ideal conditions.

8.1 REFLECTION

For a mirror reflective surface, the reflection formula is

$$\mathbf{r} = \mathbf{i} - 2(\mathbf{i} \cdot \mathbf{n})\mathbf{n}, \tag{8.1}$$

where **i** is the incoming ray direction, **n** the surface normal, and **r** the reflection direction. See Figure 8-1. The normal direction is expected to be normalized (equal to a length of one) in this form. The normal can point in either direction (e.g., upward or downward) and the formula still works. The incoming ray direction does not have to be normalized, and the outgoing direction vector will be computed having the same length as the incoming one.

The pseudocode, available at the book's source code website, is simply:

```
1 void ReflectionDirection(Vector I, Vector N, Vector & R)
2 {
3     R = I - 2*VecDot(I,N)*N;
4 }
```

8.2 REFRACTION

Light hitting water, glass, or other transparent material can reflect off the surface or can travel into the medium along the refraction direction. These materials have an additional factor that determines how they affect the ray, the *index of refraction* or *refractive index* [6]. In physical terms, light's speed

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_8

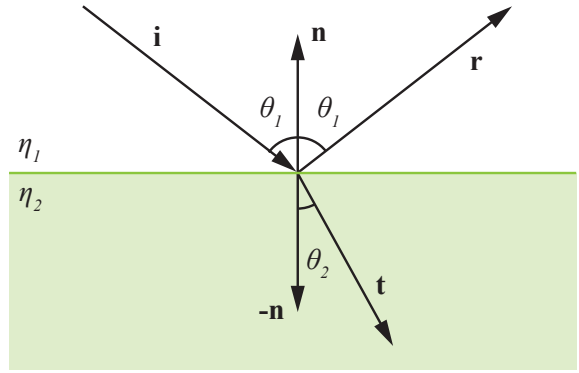


Figure 8-1. *Reflection and refraction terms.*

through a material is found by dividing by that material's index of refraction. This value for a material typically varies with wavelength, which is why prisms work and why we see rainbows. Temperature and pressure can also affect a material's refractive index, such as seen with heat haze.

In rendering, we often use the simplification that the index of refraction is held constant across wavelengths, so that there is just one refraction ray direction. Refractive index measurements are standardized as being taken at the center of the yellow "sodium doublet" wavelength, about 589 nanometers, under normal environmental conditions. The index of refraction is defined as exactly 1.0 for a vacuum. It is a tiny bit higher than 1.0 for air and other gases. Water's is 1.333, window glass's is 1.52, and diamond's is 2.417 [3, 5]. The index of refraction is usually denoted by the Greek letter η, pronounced "eta."

Snell's law [3] states that when light travels from one medium to another, the sine of the angle of incidence θ_1 and sine of the angle of refraction θ_2 are related to their respective refractive indices η_1 and η_2 by

$$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2. \tag{8.2}$$

For example, crossing from air to water, η_1 is essentially 1.0 and η_2 is 1.333. If the incoming ray's angle θ_1 is at, say, 60 degrees, the outgoing angle θ_2 is then arcsin(sin(60)/1.333), or 40.51 degrees.

As another example, if a ray travels from inside the glass wall of an aquarium into the water enclosed, η_1 is 1.52 and η_2 is 1.333. Crossing from a medium

with a high index of refraction to that of a lower one, *total internal reflection* [3, 7] can occur, in which no transmission takes place:

$$\frac{\eta_1}{\eta_2} \sin\theta_1 > 1. \tag{8.3}$$

This equation can be reorganized to compute the *critical angle*,

$$\theta_1 = \arcsin(\eta_2/\eta_1). \tag{8.4}$$

For our glass to water example, total internal reflection then occurs above an incidence angle of arcsin(1.333/1.52), about 61.28 degrees.

Because of this possibility, the following pseudocode computes a normalized refraction direction `T` and returns `true` only when total internal reflection does not occur. It assumes that the incoming ray direction `I` is also normalized. Note that `n1` stands for η_1 and `n2` for η_2.

```
bool TransmissionDirection(float n1, float n2,
        Vector I, Vector N, Vector & T)
{
    float eta = n1/n2;          /* relative index of refraction */
    float c1 = -VecDot(I, N);    /* cos(theta1) */
    float w = eta*c1;
    float c2m = (w-eta)*(w+eta); /* cos^2(theta2) - 1 */
    if (c2m < -1.0f)
        return false;            /* total internal reflection */
    T = eta*I + (w-sqrt(1.0f+c2m))*N;
    return true;
}
```

This is one of Bec's methods [1], which is efficient when ray directions are expected to be normalized. Other comparable formulas include Whitted's [4], which does not assume the incoming direction is normalized, and Heckbert's two methods [2], the second of which has fewer overall operations by using more divisions. Heckbert [2] and Pharr et al. [3] provide derivations.

The strength of the contributions from reflection and refraction rays in shading the surface is the subject of Chapter 9.

REFERENCES

[1] Bec, X. Faster refraction formula, and transmission color filtering. *Ray Tracing News*, 10(1), 1997. http://www.raytracingnews.org/rtnv10n1.html.

[2] A. S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press Ltd., 1989. http://bit.ly/AITRT.

[3] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, third edition, 2016. http://www.pbr-book.org/.

[4] Whitted, T. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980. DOI: 10.1145/358876.358882.

[5] Wikipedia. List of refractive indices. https://en.wikipedia.org/wiki/List_of_refractive_indices. Last accessed January 26, 2021.

[6] Wikipedia. Refractive index. https://en.wikipedia.org/wiki/Refractive_index. Last accessed January 26, 2021.

[7] Wikipedia. Total internal reflection. https://en.wikipedia.org/wiki/Total_internal_reflection. Last accessed January 26, 2021.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 9

THE SCHLICK FRESNEL APPROXIMATION

Zander Majercik
NVIDIA

ABSTRACT

Reflection and refraction magnitudes are modeled using an approximation of the Fresnel reflectance equations called the Schlick approximation. This chapter presents the Schlick approximation, analyzes its error when compared to the full Fresnel reflectance equations, and motivates its use in ray tracing dielectric (non-conducting) materials. It further discusses the Schlick approximation in the context of metallic surfaces and shows a possible extension for the Schlick approximation to more accurately model reflectance of metals.

9.1 INTRODUCTION

When a ray hits a surface during ray tracing, we need to compute how much light should be transmitted back along it (toward the camera for a primary ray or toward the earlier path vertex for a secondary ray). Light transmitted back along a ray hitting a surface is either reflected or refracted from that surface. Some materials, like glass, will both reflect *and* refract light. For such materials, a portion of the energy of an incoming light ray is reflected—the rest is refracted.

9.2 THE FRESNEL EQUATIONS

Reflection and refraction magnitude vary with a material's refractive index[1] η and the angle of the incoming light θ_i (the incoming angle relative to the surface normal). The portion of light reflected from a ray traveling through a material with refractive index η_1 that hits a material with refractive index η_2 at

[1] A material's *refractive index* is the ratio of the speed of light in a vacuum to the speed of light in that material. For example, water has a refractive index of $\eta = 1.333$.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_9

angle θ_i is given by the Fresnel equations:

$$R_s = \left| \frac{\eta_1 \cos\theta_i - \eta_2 \sqrt{1 - \left(\frac{\eta_1}{\eta_2} \sin\theta_i\right)^2}}{\eta_1 \cos\theta_i + \eta_2 \sqrt{1 - \left(\frac{\eta_1}{\eta_2} \sin\theta_i\right)^2}} \right|^2, \tag{9.1}$$

$$R_p = \left| \frac{\eta_1 \sqrt{1 - \left(\frac{\eta_1}{\eta_2} \sin\theta_i\right)^2} - \eta_2 \cos\theta_i}{\eta_1 \sqrt{1 - \left(\frac{\eta_1}{\eta_2} \sin\theta_i\right)^2} + \eta_2 \cos\theta_i} \right|^2. \tag{9.2}$$

The *s* and *p* subscripts denote the polarization of light: *s* is perpendicular to the propagation direction and *p* is parallel. Most ray tracers ignore light polarization by simply averaging the two equations to arrive at a final reflection magnitude $R = (R_s + R_p)/2$.

9.3 THE SCHLICK APPROXIMATION

The Fresnel equations are exact, but complicated to evaluate. In computer graphics, a simpler approximation, called Schlick's approximation, is often used instead. Schlick's approximation is given by

$$R(\theta_i) = R_0 + (1 - R_0)(1 - \cos\theta_i)^5, \tag{9.3}$$

where θ_i is again the angle of the incident ray and R_0 is the reflectivity of the material at normal incidence (you can check this by substituting $\theta_i = 0$). Here is for the Schlick approximation:

```
/** Compute reflectance, given the cosine of the angle of incidence and the
      reflectance at normal incidence. */
vec3 schlickFresnel(vec3 R0, float cos_i) {
    return lerp(R0, vec3(1.0f), pow((1.0f - cos_i), 5.0f));
}
```

The Schlick approximation is much faster to evaluate than the full Fresnel equations. In an optimized implementation, the Schlick approximation can be 32 times faster with less than 1% average error [4]. Further, the Schlick approximation only depends on the reflectance at normal incidence (R_0), which is known for many materials, whereas the Fresnel equations depend on the refractive indices η_1 and η_2.[2]

[2] Magnitude R_0 can also be computed from refractive indices like so: $R_0 = ((\eta_1 - \eta_2)/(\eta_1 + \eta_2))^2$. This makes it possible to use the Schlick approximation for reflectance at the interface of arbitrary materials. For the most common case of a material boundary with air, only reflectance at normal incidence is required.

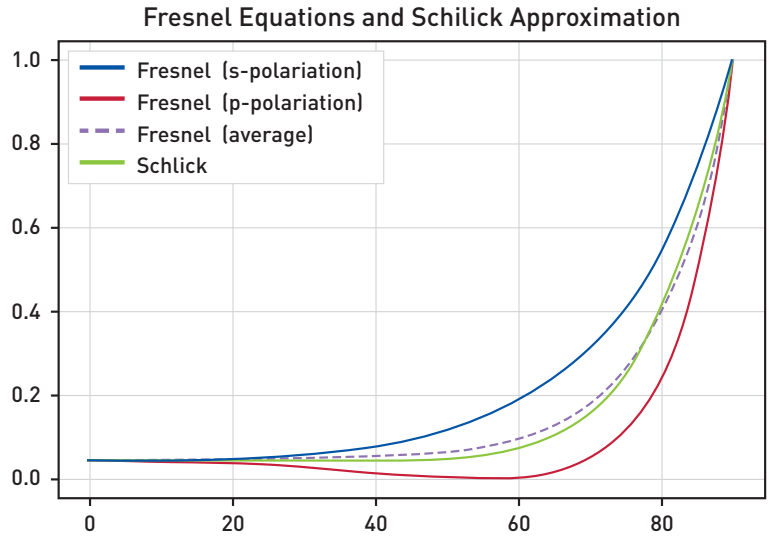


Figure 9-1. *Graph of the Fresnel equations for s-polarization (blue), p-polarization (red), the average of s- and p-polarization (dashed purple), and the Schlick approximation (green). The x-axis shows the angle of incident light (θ_i, in degrees), and the y-axis shows the portion (0–1) of incident light reflected. The Schlick approximation closely matches the averaged Fresnel equations. The Fresnel equations were computed with $\eta_1 = 1$ and $\eta_2 = 1.5$ (simulating the interface of air and glass, with incident light coming from the air). Following these, the Schlick approximation used* 0.04 *for the Fresnel reflectance at normal incidence.*

Figure 9-1 shows the magnitude of reflected radiance as a function of incident angle for a light ray traveling through air ($\eta = 1.0$) and striking glass ($\eta = 1.5$). The reflectance of *s*-polarization, *p*-polarization, their average, and the Schlick approximation are shown. Relative to the averaged Fresnel equations, the Schlick approximation gives less than 1% average error with a maximum error of approximately 3.6% at an incident angle of 85°.

The key insight is that averaging R_s and R_p already introduces an approximation to the polarized reflectance. This approximation can have high error, especially if incoming light is unevenly polarized (see Wolff and Kurlander [5] for a classic discussion of polarization in ray tracing). Given that this is already an approximation, it makes sense to fit the approximate curve with a simple polynomial instead of evaluating the full Fresnel and taking the average. The Schlick approximation achieves this.

9.4 DIELECTRICS VS. CONDUCTORS

So far when talking about reflective/refractive materials, we have been talking about *dielectrics*—materials that do not conduct electricity, such as water, wood, or stone. Materials that *do* conduct electricity, such as metals, are called *conductors*. Conductors require a more advanced form of the Fresnel equations with an additional parameter called the *extinction coefficient* (κ). The extinction coefficient represents the attenuation (amplitude reduction) of light in a volume—a lower extinction coefficient means that a material is less conductive.

The reflectance of a ray traveling through a dielectric material with refractive index η_d and striking a conductor with refractive index η_c and extinction coefficient κ at incident angle θ_i is given by

$$R_s = \frac{a^2 + b^2 - 2a\cos\theta_i + \cos^2\theta_i}{a^2 + b^2 + 2a\cos\theta_i + \cos^2\theta_i}, \tag{9.4}$$

$$R_p = R_s \frac{a^2 + b^2 - 2a\sin\theta_i\tan\theta_i + \sin^2\theta_i\tan^2\theta_i}{a^2 + b^2 + 2a\sin\theta_i\tan\theta_i + \sin^2\theta_i\tan^2\theta_i}, \tag{9.5}$$

with a^2 and b^2 given by

$$a^2 = \frac{1}{2\eta_d^2}\left\{\sqrt{\left(\eta_c^2 - \kappa^2 - \eta_d^2\sin^2\theta_i\right)^2 + 4\eta_c^2\kappa^2} + \left(\eta_c^2 - \kappa^2 - \eta_d^2\sin^2\theta_i\right)\right\}, \tag{9.6}$$

$$b^2 = \frac{1}{2\eta_d^2}\left\{\sqrt{\left(\eta_c^2 - \kappa^2 - \eta_d^2\sin^2\theta_i\right)^2 + 4\eta_c^2\kappa^2} - \left(\eta_c^2 - \kappa^2 - \eta_d^2\sin^2\theta_i\right)\right\}. \tag{9.7}$$

With extinction coefficient κ set to 0, these equations simplify to the dielectric Fresnel equations (Equations 9.1 and 9.2).

9.5 APPROXIMATIONS FOR MODELING THE REFLECTANCE OF METALS

The complex Fresnel equations are even more complicated and expensive to evaluate than the original Fresnel equations. Further, when using the complex Fresnel equations in practice, it is difficult to gain the benefit of physical correctness that should come with the higher evaluation cost (see Hoffman's discussion of these issues [2]).

For these reasons, we would prefer to continue using the Schlick approximation (or another approximation) for conductors just as we used it for

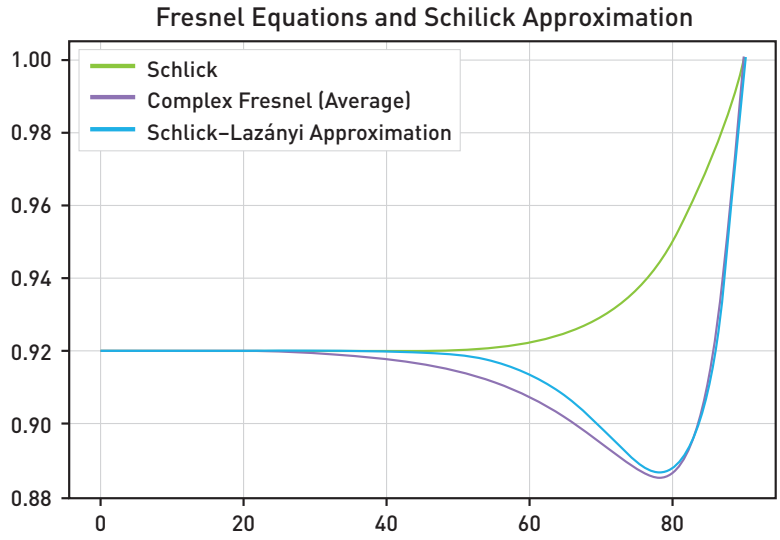


Figure 9-2. *Reflectance of aluminum at 450 nm under the complex Fresnel equations (magenta), the Schlick approximation (green), and the re-parameterized Schlick–Lazányi approximation (cyan). At a wavelength of 450 nm for incident light, aluminum has refractive index η = 0.61722 and extinction coefficient κ = 5.3031. The Schlick–Lazányi approximation was computed using α = 6 and a = 1.136 (as in Hoffman [2]). Note that because the graph shows data for a specific wavelength, a is scalar valued.*

dielectrics. The simplest option that works well in practice is to use an RGB color value for R_0 (instead of the scalar value presented in Equation 9.3) to approximate different reflection behavior along the visible spectrum. Further, using an RGB color value allows one to use the same material shader to evaluate reflection magnitude for both metals and dielectrics.

Even when using an RGB color, the Schlick approximation can still show significant error when evaluated for conductors, especially at glancing angles (see Gulbrandsen [1] for a parameterization that addresses glancing angles specifically). This error can be decreased using the Lazányi–Schlick approximation [3], which adds an error term to the RGB Schlick approximation to account for metals:

$$R(\theta_i) = R_0 + (1 - R_0)(1 - \cos\theta_i)^5 - a\cos\theta_i(1 - \cos\theta_i)^\alpha, \tag{9.8}$$

where R_0 is the RGB reflectance at normal incidence and a and α are configurable parameters.[3] It can be used with a default parameter of α = 6.

[3] Equation 9.8 presents the error term alongside the RGB Schlick approximation, matching Hoffman [2], and so differs slightly from the original paper by Lazányi and Szirmay-Kalos [3], which presents a Schlick approximation parameterized by η and κ.

The parameter *a* is computed per material based on the Fresnel reflectance of the material at the angle where the Lazányi error term is highest. For the exact equation for *a* and the derivation of the default value for *alpha*, see Hoffman [2]. Here is code for the Lazányi–Schlick approximation:

```
/** Return reflectance, given the cosine of the angle of incidence and the
    reflectance at normal incidence. */
vec3 schlickLazanyi(vec3 R0, float cos_i, vec3 a, float alpha) {
    return schlickFresnel(R0, cos_i) -
           a * cos_i * pow(1 - cos_i, alpha);
}
```

Figure 9-2 shows the reflectance curve for aluminum at a wavelength of 450 nm under the complex Fresnel equations, the basic Schlick approximation, and the re-parameterized Schlick–Lazányi approximation. Compared to the complex Fresnel equations, the Schlick approximation gives an average error of 1.5%, with a maximum error of 6.6%. The re-parameterized Schlick–Lazányi approximation is much closer with an average error of 0.22%, with a maximum error of 0.65%.

REFERENCES

[1] Gulbrandsen, O. Artist friendly metallic Fresnel. *Journal of Computer Graphics Techniques*, 3(4):64–72, 2014. http://jcgt.org/published/0003/04/03/.

[2] Hoffman, N. Fresnel equations considered harmful. In *Eurographics Workshop on Material Appearance Modeling*, pages 7–11, 2019. DOI: 10.2312/mam.20191305.

[3] Lazányi, I. and Szirmay-Kalos, L. Fresnel term approximations for metals. In *World Society for Computer Graphics*, pages 77–80, Jan. 2005.

[4] Schlick, C. An inexpensive BRDF model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246, 1994. DOI: 10.1111/1467-8659.1330233.

[5] Wolff, L. B. and Kurlander, D. J. Ray tracing with polarization parameters. *IEEE Computer Graphics and Applications*, 10(6):44–55, 1990. DOI: 10.1109/38.62695.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 10

REFRACTION RAY CONES FOR TEXTURE LEVEL OF DETAIL

Jakub Boksansky, Cyril Crassin, and Tomas Akenine-Möller
NVIDIA

ABSTRACT

Texture filtering is an important implementation detail of every rendering system. Its purpose is to achieve high-quality rendering of textured surfaces, while avoiding artifacts, such as aliasing, Moiré patterns, and unnecessary overblur. In this chapter, we extend the ray cone method for texture level of detail so that it also can handle refraction. Our method is suitable for current game engines and further bridges the gap between offline rendering and real-time ray tracing.

10.1 INTRODUCTION

Accurate rendering of transparent and semitransparent surfaces is one of the many appealing features of renderers based on ray tracing. By simply following the paths of rays refracted on transmissive surfaces, it is possible to render materials, such as glass and water, with great realism. Ray tracing enables us to take refraction indices of materials on both sides of the surface into account and "bend" the rays in a physically correct way, which is an effect that is difficult to achieve using rasterization and which contributes significantly to the realistic rendering of semitransparent materials. Here, we focus on using the ray cone method [1, 2] for the filtering of textures on surfaces seen by refracted rays to achieve the same level of visual quality as for reflected and primary rays. An example is shown in Figure 10-1.

The geometry of perfect refraction (and reflection) has been covered in detail in Chapters 8 and 9 and by de Greve [5], including the interesting case of total internal reflection and the relation of Fresnel equations to the modeling of transmissive surfaces. A practical implementation in a ray tracer is covered in Chapter 11.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_10



Figure 10-1. *Refractive materials, such glass and water, can be rendered accurately using ray tracing with respect to given indices of refraction. This is a significant improvement compared to rasterization. This picture was rendered using a fast path tracer, which correctly integrates over all necessary domains in order to reproduce refraction effects as well as camera defocus blur.*

A more complex case is the modeling of refractions on rough surfaces, e.g., frosted glass and ice, where rays are not refracted perfectly, but their direction is randomized. A model for such materials is described by a bidirectional transmittance distribution function (BTDF). A formal formulation of such a BTDF employing a widely used Cook–Torrance microfacet model was provided in the seminal paper by Stam [9], and importance sampling was added by Walter et al. [10]. A more efficient importance sampling method was later developed by Heitz [7].

Though the problem of ray cone refraction might at first seem to be the same as reflection, the fundamental difference lies in the fact that indices of refraction are typically different for bodies at opposing sides of the hit point surface, whereas they are always the same for reflecting ray cones. This introduces the relative index of refraction η, which has significant impact on the refracted ray direction. It is not only the width of the cone that changes, but also its geometry. Depending on the relative index of refraction (whether a ray refracts from an optically denser to a thinner environment or vice versa), the cones can either shrink or grow. Also, the centerline of the refracted cone can differ significantly from the direction of the refracted ray. Note that altering the refracted ray direction to reflect this change is not viable, in our

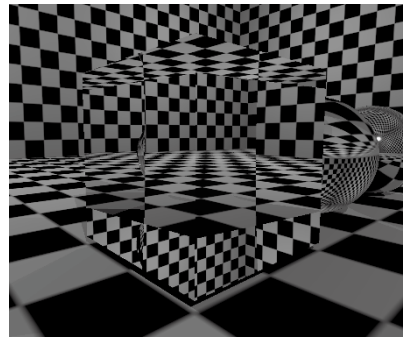


Figure 10-2. *A glass cube showing refraction in the center and TIR closer to the sides.*

opinion, as it could miss geometry that would be hit under certain angles, essentially sampling a correct footprint from incorrect geometry. Such a solution would be more expensive as well.

When the ray cone refracts from an optically denser to a thinner environment, total internal reflection (TIR) can occur. A good real life example is *Snell's window* visible when an observer underwater looks up to the surface. Another example is shown on the glass cube in Figure 10-2. For incident rays making an angle with a normal that is larger than a certain critical angle, a perfect reflection occurs and we can use the ray cone solution for reflected rays. In those cases, no refraction occurs. A problem occurs on the boundary between a fully refracted and a totally internally reflected (TIRed) ray, when the ray is incident under an angle close to the critical angle. In this case, one part of the ray cone is reflected and other part is TIRed (see Figure 10-3).

The method presented in this chapter calculates an approximate refracted ray cone that attempts to match the footprint of a "perfectly refracted" cone as closely as possible without altering the refracted ray direction. As in other methods [3, 6, 8], we also only follow the part of the cone where the central ray ends up; i.e., if it is refracted, we construct a refracted ray cone, and if the central ray is TIRed, we construct a reflected ray cone.

10.2 OUR METHOD

A ray cone is defined by a width w, a spread angle α, a ray origin O, and a direction $\mathbf{d}$ [2], as illustrated in Figure 10-4, left. The curvature approximation at the hit point is modeled as a signed angle β, where a positive sign indicates

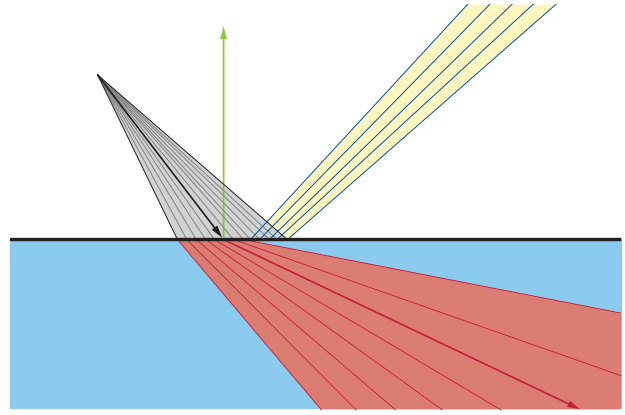


Figure 10-3. *A ray cone (in gray) is transported via a surface interface from a thicker medium to a thinner one. The central ray of the ray cone is refracted into the red arrow. Note, however, that if the ray cone is seen as a set of internal rays, four of these will be totally internally reflected, which forms the yellow part. The rest are refracted into the red volume.*

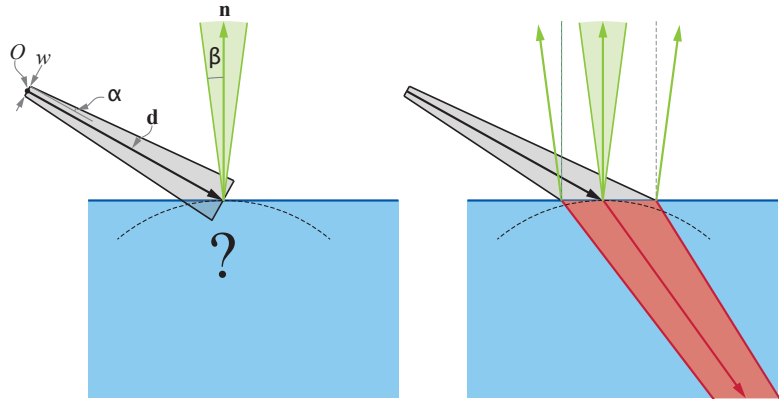


Figure 10-4. *Left: a gray ray cone, defined by an origin O, a direction* **d**, *a width w, and a half cone angle α. The spread of the normal* **n** *is modeled by the angle β. Right: we create our refraction ray cone approximation from the red volume. Dashed lines indicate the curvature of the surface at the hit point.*

a convex surface and a negative sign indicates a concave surface. Our assumption is that a reasonable approximation will be obtained by handling the computations in two dimensions, by tracking the lower and upper parts of the ray cone, and by refracting those with the perturbed normals as shown in Figure 10-4, right.

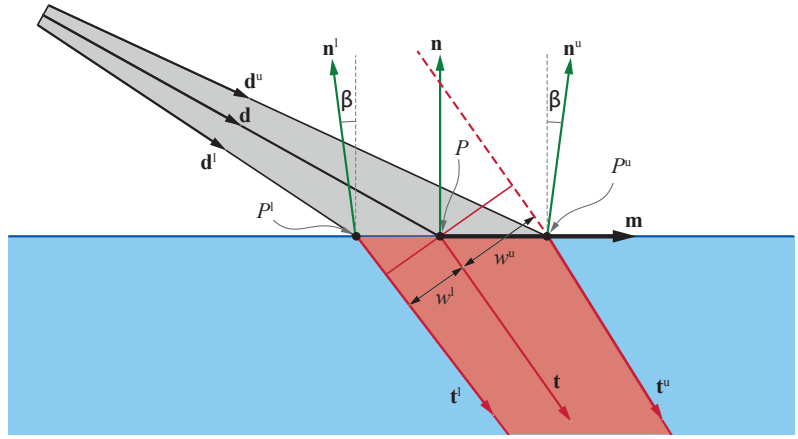


Figure 10-5. *The geometric setup used in the derivation of the refracted ray cone.*

In the following, all references are made to Figure 10-5. We use the hit point P as the origin of the two-dimensional coordinate frame, and we use $\mathbf{n}$ as the y-axis and $\mathbf{m}$ as the x-axis. The vector $\mathbf{m}$ needs to be orthogonal to $\mathbf{n}$ and also parallel to $\mathbf{d}$, and this can be done with a projection and normalization as

$$\mathbf{m} = \frac{\mathbf{d} - (\mathbf{n} \cdot \mathbf{d})\mathbf{n}}{\|\mathbf{d} - (\mathbf{n} \cdot \mathbf{d})\mathbf{n}\|}. \tag{10.1}$$

We assume that all vectors and points are in this two-dimensional frame. We create the *upper* direction vector $\mathbf{d}^u$ by rotating $\mathbf{d}$ by $+\alpha$, and the *lower* direction vector $\mathbf{d}^l$ by rotating $\mathbf{d}$ by $-\alpha$. The upper ray, with direction $\mathbf{d}^u$, starts from the origin O offset by $w/2$ in the direction orthogonal to $\mathbf{d}$, and similar for the lower ray. These are then intersected with the x-axis, which gives us P^u and P^l. This is illustrated in Figure 10-5.

We use an augmented refraction function that returns a vector located in the plane being spanned by the incident ray direction and the normal, and also orthogonal to the normal, if the ray is TIRed. Otherwise, standard refraction is used. For example, if $\mathbf{d}$ would have been TIRed in Figure 10-5, then our refraction function would return $\mathbf{m}$. From the spread of the normal and the normal itself, $\mathbf{n}$, we also create $\mathbf{n}^u$ and $\mathbf{n}^l$, which are shown in Figure 10-5. The refracted direction $\mathbf{t}$ is computed by calling the refraction function with $\mathbf{n}$ and $\mathbf{d}$ and the ratio of indices of refraction. This is done analogously to create $\mathbf{t}^u$ and $\mathbf{t}^l$ using $\mathbf{n}^u$ and $\mathbf{n}^l$.

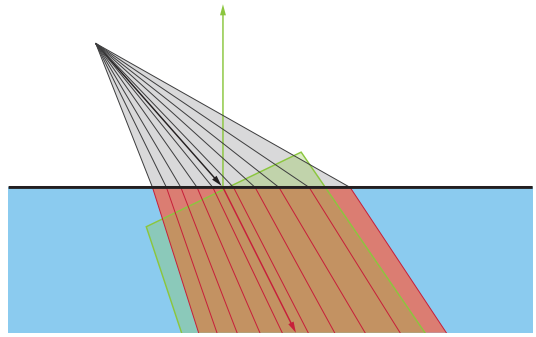


Figure 10-6. *A refracted cone calculated using our method (light green) and the ground truth approximated using many rays (red).*

Given the computations above, the refracted ray cone is constructed as follows. The refracted ray cone origin is P and its direction is $\mathbf{t}$. The width w of the refracted ray cone is computed as

$$w = w^{\mathrm{u}} + w^{\mathrm{l}}, \tag{10.2}$$

where w^{u} and w^{l} are the widths shown in Figure 10-5. The width w^{u} is computed as the length along a direction that is orthogonal to $\mathbf{t}$ from the origin to the line defined by $\mathbf{t}^{\mathrm{u}}$ and the hit point P^{u}, and analogously for w^{l}. The half cone angle α of the refracted ray cone is computed as half the angle between $\mathbf{t}^{\mathrm{u}}$ and $\mathbf{t}^{\mathrm{l}}$ together with a sign as

$$\alpha = \frac{1}{2} \arccos\left(\mathbf{t}^{\mathrm{u}} \cdot \mathbf{t}^{\mathrm{l}}\right) \operatorname{sign}\left(t_x^{\mathrm{u}} t_y^{\mathrm{l}} - t_y^{\mathrm{u}} t_x^{\mathrm{l}}\right). \tag{10.3}$$

In order to further improve the quality of our method, we suggest using higher-quality anisotropic filtering [1] for hits after refraction has occurred. The ray cone geometry changes significantly after refracting, as described in Section 10.1 and illustrated in Figure 10-6. To compensate for the imperfect approximation of the refracted cone, we can use the anisotropic filter to at least better match the elliptical footprint at the hit point.

One situation that is not explicitly handled in our method is when the incident ray is below the perturbed normal of the surface. In this case, a reflection should occur instead of refraction. However, during our experiments, we did not detect this situation happening, and without handling it explicitly, our implementation causes the cone to grow instead of reflecting. To handle this situation, we propose to clamp incident vectors to be at most 90° away from the normal.

Our method cannot only be used for perfect refraction, but also for rough refractions when, e.g., a microfacet-based BTDF is used to generate a randomized refracted direction based on surface roughness. In this case, the half-vector used for refracting the incident direction should be used as the normal for our method because it is, in fact, the normal of the microfacet that is used to refract the ray. Methods for stochastic evaluation of microfacet BTDF typically generate these half-vectors, which can be used directly. Alternatively, we can calculate such a half-vector using

$$\mathbf{n} = -\frac{\eta\mathbf{t} + \mathbf{d}}{\|\eta\mathbf{t} + \mathbf{d}\|}. \tag{10.4}$$

10.3 RESULTS

To evaluate our method, we compare the results to the ground-truth reference that stochastically samples the correct footprint and to the method based on ray differentials [8]. In Figure 10-7, we show results from a Whitted ray tracer with several different texture filtering methods, namely, accessing only mipmap level 0 (no filtering), using ray cones with isotropic filtering [2], using ray cones with anisotropic filtering [1], and using ray differentials with both isotropic and anisotropic filtering [8].

Our test scene contains a reflective sphere, a solid glass cube, and a glass sphere, where the latter two cover refraction under many angles in one view and also the case of total internal reflection. However, none of the methods explicitly handle the boundary between refraction and TIR (which is usually one or two pixels wide), and only the ground-truth method can render it correctly—the other methods produce a boundary that is too sharp. Visual results of our ray cone method are mostly comparable to the ray differentials (see Figure 10-7). Our method, however, requires less per-ray storage and overhead. Note that, as mentioned previously, we prefer to use anisotropic filtering for refraction, which also gives results closer to the ground truth.

Performance of our method is nontrivial to evaluate, as opposed to when only reflection occurs, as our new method is only used for rays hitting transmissive materials. We used a test scene containing glass objects and changed the view so that most of the scene is seen through the lens. Results are summarized in Figure 10-8. Performance of all methods is near identical, but slower than simply accessing mip level 0, which is expected. (The rendering times for the viewpoint shown in Figure 10-1 with path tracing at 4 spp are 76.8 ms for mip level 0, 75.4 ms for ray cones with isotropic filtering,

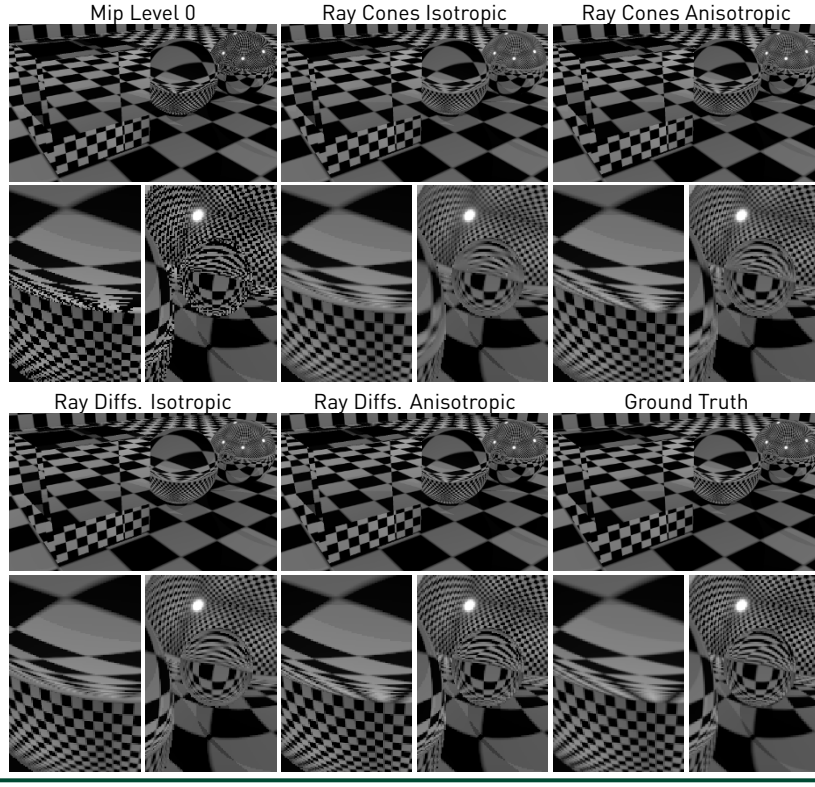


Figure 10-7. *Results for our refracted ray cone method, which used the Unified method [1] for computing surface spread. The ground truth was generated by stochastically sampling the direction inside the screen-space ray cone through a pixel with 10,000 samples per pixel. Note that anisotropic ray cones generate results that are close to anisotropic ray differentials in the left zoom-in, whereas in the right zoom-in, the ray cones are more blurry. This is because ray cones can only model circular cones and this situation requires more flexibility than that. Ray differentials have a more expensive representation, which allows for better results.*

82.7 ms for ray differentials with isotropic filtering, and 84.9 ms for ray differentials with anisotropic filtering.) The isotropic versions are faster than the anisotropic ones, but the method when we only use anisotropic filtering after a refraction event has occurred can be expected to be faster than anisotropic filtering for all hits in demanding scenes.

We also validated our approach inside a path tracer that stochastically generates path samples that integrate over both the camera pixel footprint (antialiasing) and a thin lens aperture (providing a defocus blur effect). We adapted the ray cone creation procedure in order to better match the *double cone* footprint of the light beams generated by such a setup. We take



Figure 10-8. *The Marbles scene rendered at* 1920 × 1080 *resolution with our path tracer. Left: for reference, the picture rendered using full-resolution textures (mip level 0). Right: our result using ray cones with isotropic filtering produces visually identical results. The viewpoint in the top row was rendered using 1000 spp (samples per pixel) for both approaches, and the bottom row uses 400 spp.*

advantage of the standard functionality of the ray cones and create them such that they start from the camera at aperture width, converge to zero at the focal distance, and then grow again. This allows our method to access mip levels with lower resolution even more in the blurry regions of the defocus blur. In such a path tracing setup, ray cones allow for saving bandwidth during texture sampling operations, resulting in ca. 1.5% full-frame performance improvement in the scene presented in Figure 10-1 and Figure 10-8. On this scene, our approach also renders ca. 10% faster than ray differentials with isotropic filtering and ca. 12% faster than ray differentials with anisotropic filtering. As it provides the path tracing integrator with prefiltered bandlimited texture samples, our approach also improves image convergence, as can be seen in the low-sample count results presented in Figure 10-9. Those pictures were rendered at interactive frame rates (62 ms per frame) after three accumulated frames of a static camera, using 1 spp per frame (resulting in 3 spp effective) and denoised using the NVIDIA OptiX denoiser.

Our tests were performed using an NVIDIA RTX 3090 GPU (Ampere). Compared to always accessing the finest mip level (0), ray cones with mipmapping can achieve speedups in general. This is due to accessing mip levels with lower resolution more often. However, speedups can be expected



Figure 10-9. *Interactive path tracing after three accumulated frames with low sample count (1 spp/frame resulting in 3 spp effective) using screen-space denoising and rendered at* 1920×1080*. Our approach (bottom insets) provides smoother outputs in the blurry regions compared to sampling textures at maximum resolution (mip level 0, top insets), which can help with providing faster convergence speeds.*

only for the most demanding scenes with a very large number of high-resolution textures. We also used uncompressed textures for those experiments, and enabling texture compression can reduce those speedups.

10.4 CONCLUSION

With the growing use of high-resolution textures and ray tracing in games, it is important to support a precise solution for their filtering in order to preserve the fine detail meticulously created by artists. In this chapter, we have continued development of texture filtering with ray cones and added support for refraction. Building on top of previous work, ray cones can now support a range of use cases including reflection, refraction, isotropic or anisotropic filtering, shading level of detail, integration with or without G-buffer, and more. Ray cones are a lightweight and relatively easy-to-integrate method, and we provide a publicly available implementation as part of the Falcor framework [4] (see the `RayFootprint` class).

REFERENCES

[1] Akenine-Möller, T., Crassin, C., Boksansky, J., Belcour, L., Panteleev, A., and Wright, O. Improved shader and texture level of detail using ray cones. *Journal of Computer Graphics Techniques (JCGT)*, 10(1):1–24, 2021. http://jcgt.org/published/0010/01/01/.

[2] Akenine-Möller, T., Nilsson, J., Andersson, M., Barré-Brisebois, C., Toth, R., and Karras, T. Texture level-of-detail strategies for real-time ray tracing. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 20, pages 321–345. Apress, 2019.

[3] Belcour, L., Yan, L.-Q., Ramamoorthi, R., and Nowrouzezahrai, D. Antialiasing complex global illumination effects in path-space. *ACM Transactions on Graphics*, 36(4):75b:1–75b:13, 2017. DOI: 10.1145/3072959.2990495.

[4] Benty, N., Yao, K.-H., Foley, T., Kaplanyan, A. S., Lavelle, C., Wyman, C., and Vijay, A. The Falcor real-time rendering framework. https://github.com/NVIDIAGameWorks/Falcor, 2017.

[5] De Greve, B. Reflections and refractions in ray tracing. https://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf, 2006.

[6] Elek, O., Bauszat, P., Ritschel, T., Magnor, M., and Seidel, H.-P. Progressive spectral ray differentials. *Computer Graphics Forum*, 33(4):113–122, Oct. 2014. DOI: 10.1111/cgf.12418.

[7] Heitz, E. Sampling the GGX distribution of visible normals. *Journal of Computer Graphics Techniques*, 7(4):1–13, 2018. http://jcgt.org/published/0007/04/01/.

[8] Igehy, H. Tracing ray differentials. In *Proceedings of SIGGRAPH 99*, pages 179–186, 1999.

[9] Stam, J. An illumination model for a skin layer bounded by rough surfaces. In *Eurographics Workshop on Rendering*, pages 39–52, 2001.

[10] Walter, B., Marschner, S. R., Li, H., and Torrance, K. E. Microfacet models for refraction through rough surfaces. In *Rendering Techniques*, pages 195–206, 2007.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 11

HANDLING TRANSLUCENCY WITH REAL-TIME RAY TRACING

Tianyi "Tanki" Zhang
NVIDIA

ABSTRACT

Translucency is a hard problem in real-time rendering. Despite the various techniques introduced to handle translucency in rasterization, the result is far from satisfactory, due to the limited geometry surface information outside of the screen space. For example, multi-layer alpha blending is nontrivial to implement and can still fail in many cases, not to mention rendering physically based translucency effects like refraction and absorption. This chapter provides an overview of translucent rendering effects with the aid of hardware-accelerated ray tracing. Ray tracing allows techniques to become possible that are not possible with rasterization, while also improving others. We are going to show how to implement transparency in ray tracing while also discussing trade-offs between correctness and accuracy. Instead of focusing on explaining different algorithms, the major focus of this chapter is implementation. This chapter will use the terminology of DirectX Raytracing (DXR) for simplicity, but the methods can also be applied for Vulkan.

11.1 CATEGORIES OF TRANSLUCENT MATERIAL

When we talk about translucency, regardless of how we define the parameterization of the material models, there are two different categories. On one hand, physically based translucent materials are modeled through physical properties, scattering coefficient σ_s and absorption coefficient σ_a, which are generally used to describe participating media. Coefficient σ_s describes the probability of an out-scattering event occurring per unit distance, and σ_a represents the probability that light is absorbed per unit distance traveled in the medium [3]. When light travels through a volume, different events happen (Figure 11-1).

In the real three-dimensional world, all objects—and the space between them—have a certain volume. We can define scattering and absorption based

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_11

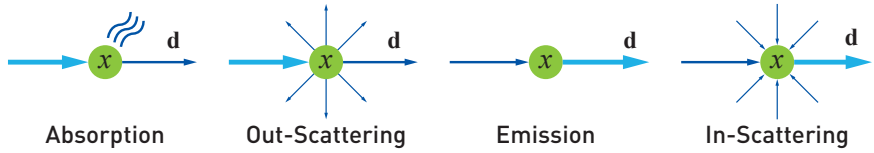


Figure 11-1. *Different events when energy travels through a participating medium [1].*

on the distance within those volumes. But in a virtual world, not all objects are created as closed geometries, which simply means that if a ray passes through the geometry, we will have pairs of intersection points, where the ray first hits the frontface and then hits the backface. When we want to render geometries without volume, for example, quads and disks, assuming a thickness can achieve physically plausible results.

On the other side, opacity-based, non-physically based materials are highly adopted in game engines. A single float opacity α is applied to indicate how "transparent" a surface is, as shown in Figure 11-2. The lower the opacity value is, the more transparent the surface is. The core idea of rendering semitransparent materials is called *alpha blending*. There are many different approaches with the rasterization pipelines, with different pros and cons [1].

11.2 OVERVIEW

Figure 11-3 is a common example that a renderer needs to handle. Shooting one ray bouncing in the scene and spawning more secondary rays along the way to make sure all sources of contribution are well sampled is the most ideal approach, but it requires much more time than a real-time renderer can afford. Thus, translucency is broken into different effects and handled

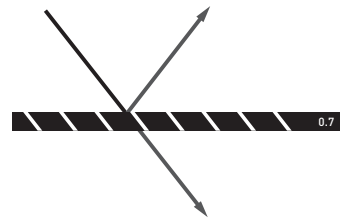


Figure 11-2. *When light hits a surface with opacity, it's easy to think about it as if some part of the energy will go through. In this example, the surface has an opacity of 0.7, which means that 70% of the energy is reflected and the remaining 30% of energy goes straight through the surface.*

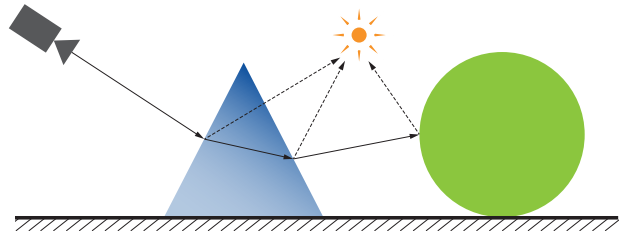


Figure 11-3. *When the ray travels through the scene, the primary ray is refracted and enters the object, then is refracted again when leaving the object and hitting an object with a scattering material. The renderer needs to evaluate the radiance at all points by sampling the light and computing the visibility term to calculate the total throughput that the ray is carrying.*

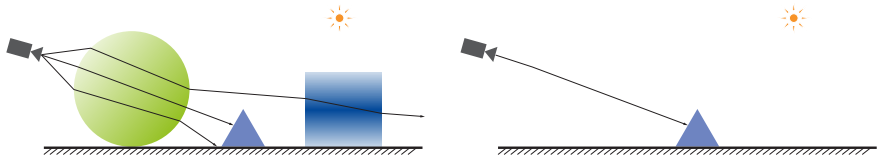
separately in different passes with different assumptions in real time and also skips the parts that only have limited contribution to the result fidelity.

To shade this example, we will start with a path tracing–style translucent pass (Section 11.3), which can handle "clear" materials well. Clear materials are those that only have absorption (or opacity) and do not have scattering. Section 11.5 introduces a setup to handle visibility for semitransparent materials, which can also be directly applied to render transparent shadows.

11.3 SINGLE TRANSLUCENT PASS

A good start to handle the translucent effects for a real-time renderer is to add a path tracing fashion translucent pass, where a primary ray is traced through the scene for each pixel, refracted upon entering or leaving the geometries until it reaches the max bounce or hits an opaque surface. Figure 11-4a illustrates how one ray travels through the scene in that pass. A primary ray starts from the camera and hits the translucent surface, where we shade the surface with incoming light radiance. Then, according to the index of refraction (IoR), the ray is refracted and enters the geometry. When the ray is inside the geometry, we can also sample the distance and phase functions to estimate single scattering events. When a ray leaves the geometry from the other side, the hit point is shaded and absorption-based transmittance is evaluate according to the Beer–Lambert law.

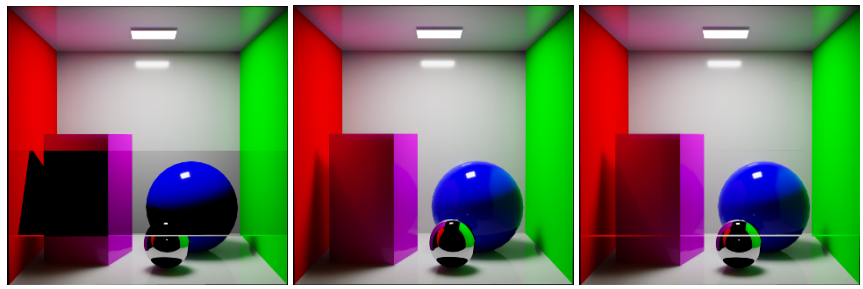
Unfortunately, the plain approach just described can cause many obvious visual artifacts in real time. A good example is when a user puts a slice of glass in front of the camera (Figure 11-5). Because opaque objects are all shaded with deferred passes before the forward translucency pass and for



(a) *Light transport in the translucent pass* **(b)** *Light transport in the opaque pass*

Figure 11-4. *(a) An example of the light transport in the translucent pass in 2D. The sphere and cube are translucent and the triangle is opaque. Only a few representative paths are drawn to keep the figure clean. (b) The main pipeline shades opaque objects only and stores the results in the G-buffer. In this example, both rays hit the same location. Instead of shading the hit point again, the renderer can immediately fetch the radiance from the G-buffer to take advantage of the evaluated result.*

performance consideration, the renderer will usually only compute basic energy contributions like direct lighting. If we want to keep the same level of fidelity for those opaque objects behind the translucent objects, the pass needs to invoke all the algorithms for different effects, which can quickly become very expensive. One trick to improve this issue is to fetch the output radiance buffer composited from the previous shading passes for opaque objects when it's possible, instead of always shading the hit points. Many different properties of the hit points can be tested against the information in the G-buffer to tell whether the ray hits something that is included in the G-buffer, including object ID, world position, normal, etc. See Listing 11-1.



(a) *Without fallback* **(b)** *Reference* **(c)** *With G-buffer fallback*

Figure 11-5. *(a) A scene with only opaque objects, where part of the view is blocked by a thin slice of glass, without fetching the G-buffer for refracted rays when the translucent rays hit the opaque objects. (b) A reference image without the glass slice. (c) When the translucent path hits an opaque object, the program checks the geometry and material properties and tries to fetch the G-buffer radiance. Notice that the "over-exposed" looking lines in the figures are not artifacts, but are the inner bottom face of the glass slice.*

Listing 11-1. *Pseudocode of a recursive path tracing–style translucent path.*

```
void TraceTranslucency(RayDesc ray, inout float3 throughput, inout float3
     radiance, int depth)
{
    if(depth > MAX_DEPTH) return;
    HitInfo hit = TraceRay(ray, ...);
    float3 directLightingOutgoingRadiance = shaderHit(hit);
        // Direct lighting
    if(hit.isBackHit) {
        throughput *= exp(-hit.absorptionCoefficient * hit.hitT);
    }
    radiance += throughput * directLightingOutgoingRadiance;

    RayDesc reflectionRay = getReflectionRay(ray, hit);
    TraceTranslucency(reflectionRay, throughput, radiance, depth + 1);
    RayDesc refractionRay = getRefractionRay(ray, hit);
    TraceTranslucency(refractionRay, throughput, radiance, depth + 1);
}

void TranslucentRayGen()
{
    uint2 pixelIndex = DispatchRaysIndex().xy;
    RayDesc ray = GetPrimaryFromPixel(pixelIndex);
    float3 totalRadiance = 0;
    TraceTranslucency(ray, float3(1, 1, 1), totalRadiance, 0);
    OutputToTexture(pixelIndex, totalRadiance);
}
```

11.4 PIPELINE SETUP

The renderer needs to have the ability to invoke a `TraceRay()` call that only traces some of the geometries in the rendering pipeline before the developer can implement the pass in Section 11.3.

There are two steps to achieve it. First, the renderer needs a way to categorize different materials and be able to tell what kind of material an object instance has. Second, the shader needs a way to access this piece of information and filter out undesired geometry. See Listing 11-2.

The `InstanceMask` field in `D3D12_RAYTRACING_INSTANCE_DESC` can be used for the first task. It is an 8-bit mask, declared when constructing geometry instances for the acceleration structure. Users can use those bits to categorize geometries. One of the arguments of the `TraceRay()` function is `InstanceInclusionMask`, and the driver will check `!((InstanceInclusionMask&InstanceMask)&0xff)` to skip certain instances. The renderer can take advantage of this feature to decide which categories are accepted, while others will be ignored during the traversal. This is the fastest way to filter geometries that also provides a fine granularity

Listing 11-2. *Essential pipeline setups.*

```
// Setup on the CPU side

///// This part should be shared between CPU and GPU. /////
#define INSTANCE_OPAQUE             (1<<0)
#define INSTANCE_TRANSLUCENT        (1<<1)
#define INSTANCE_IS_SOMETHING_ELSE  (1<<2)
//----------------------end shared code---------------------//

// CPU
D3D12_RAYTRACING_INSTANCE_DESC opaqueInstance, translucentInstance;
opaqueInstance.InstanceMask |= INSTANCE_OPAQUE;
translucentInstance.InstanceMask |= INSTANCE_TRANSLUCENT;
// ...

// GPU
// Only trace opaque geometries.
uint rayMask = INSTANCE_OPAQUE;
// Trace opaque and translucent geometries.
// uint rayMask = INSTANCE_OPAQUE | INSTANCE_TRANSLUCENT;

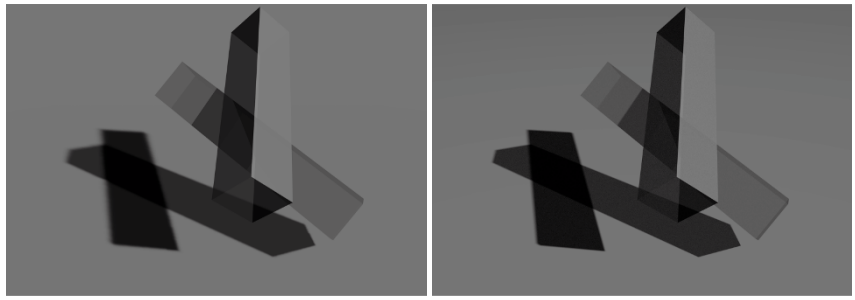
TraceRay(sceneAcc, flags, rayMasks, /*other arguments*/);
```

to apply control at the per-ray level. The only con is that there are only 8 bits available, which can be quickly consumed in a production renderer.

The renderer can bind per geometry data as part of the local root signature for each hit shader. Another way to store category information is to deploy it as part of the geometry data. During the runtime, the user can either pass the categories as payload or configure it as some global constant data bound to the pass. By conditionally invoking `IgnoreHit()` in the any-hit shader, the renderer will be able to filter out geometries with the overhead of any hit invocation. This approach provides the max system flexibility with the cost of increased overhead. Although developers can shave off some of the overhead by enforcing a `RAY_FLAG_FORCE_OPAQUE` ray flag, closest-hit shader invocation can still cause large performance penalties if instances invoke many different ones due to frequent instruction cache misses. Besides, it sacrifices the chance to handle extra logic through any-hit shaders.

11.5 VISIBILITY FOR SEMITRANSPARENT MATERIALS

We are going to describe transparent shadows that allow objects to partially block light (as in Figure 11-6), as opposed to traditional shadows where objects either fully block light or fully let light pass. Because of this, our visibility term will not be a binary 0 or 1 value but instead will be continuous



(a) *Semitransparent material in real time* **(b)** *Path tracing reference*

Figure 11-6. *Two transparent boxes overlapped with each other. One of the major challenges is to correctly handle the visibility term, including transparent shadows that fall onto the transparent geometry.*

floating-point values between 0 and 1. This allows semitransparent objects to cast colorful shadows in the renderer, increasing realism and allowing artistic effects that go beyond what rasterized monochrome shadow maps can give us.

The attenuation of visibility, which is transmittance in the real world, happens when the light passes through some volume so that part of the energy is absorbed. Because semitransparent materials try to describe opacity without volume, one way to evaluate the visibility term is by blending the color with the opacity. Regardless of the material models, let's say that the surface has opacity α. Then, the visibility can be represented as $(1 - \alpha)$. Notice that although α usually denotes a single floating-point number, it can also be three float numbers, each of which represents the opacity of a single channel of RGB color.

Given some objects' opacity values as $\alpha_0, \alpha_1, \alpha_2, \alpha_3, \ldots \alpha_n$, the overall visibility v is

$$v = 1 - \prod_{i=1}^{n}(1 - \alpha_i) \tag{11.1}$$

Unlike handling alpha blending for rendering transparent objects, as shown in Figure 11-7, notice that $1 - v$, which is the overall opacity α, is order independent. This is a good property for implementation because the renderer can take advantage of an any-hit shader, which can be invoked in arbitrary order along the ray, instead of casting rays one after another to process them from front to back. Listing 11-3 shows the setup of computing the visibility. The ray payload stores the overall visibility v, which is initialized

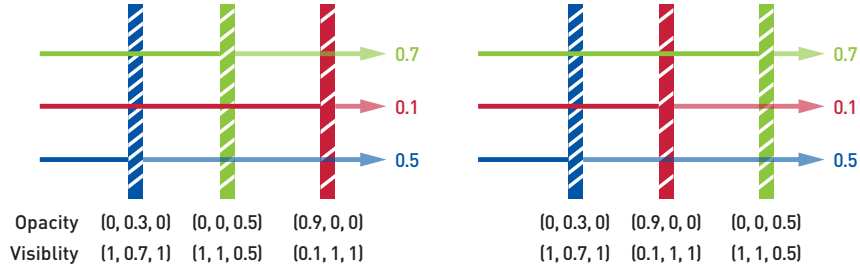


Figure 11-7. *The color of the transparent shadow depends on how much energy from the light source is left. The result is order independent. In this example, for demonstration, the light ray is illustrated by three color beams, representing the R, G, and B channels. When going through the surface, the energy attenuates according to the surface opacity.*

Listing 11-3. *Visibility ray setup: payload and raygen shader.*

```
struct Payload
{
    float3 accumulatedVisibility;
    float hitT;
};
struct HitInfo;
float3 TraceVisibilityRay(HitInfo hit)
{
    Payload payload;
    payload.accumulatedVisibility = float3(1, 1, 1);
    payload.hitT = 0;
    RayDesc shadowRay = CreateShadowRayFromHit(hit);
    // It is fine to use RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH.
    TraceRay(shadowRay, ..., RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH, ...,
        payload);
    return payload.accumulatedVisibility;
}
```

to 1. It is worth mentioning that `RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH` still works with this setup—as soon as the visibility reduces to 0 or the ray hits any opaque objects, the traversal is terminated. Without it, the ray would just keep testing against transparent hits, even if they were behind an opaque object that the ray had hit. Another alternative to improve the performance is to limit the max number of translucent hits with the trade-off of some artifacts.

We evaluate the material properties to get opacity α_i, and $v = v * (1 - \alpha_i)$. As long as the new v is greater than 0, we invoke `IgnoreHit()` but still update the hit information to make sure that we still get the correct closest-hit information through the payload (Listing 11-4). This setup can bring two

Listing 11-4. *Visibility ray setup: any-hit shader.*

```
[shader("anyhit")]
void ahs(Payload payload)
{
    // geometryData is constant data that bound to the shader with
    // geometry instance data.
    // Each any-hit shader includes the corresponding material shader logic.
    float3 opacity = materialOpacity(geometryData);
    float3 visibility = float3(1, 1, 1) - opacity;
    payload.accumulatedVisibility *= visibility;
    if(payload.hitT == 0) {
        payload.hitT = RayCurrent();
    } else {
        payload.hitT = min(payload.hitT, RayCurrent());
    }
    if(any(payload.accumulatedVisibility > 0)) {
        IgnoreHit();
    }
}
```

potential outcomes. In some cases, if there is nothing along the ray or all hits are ignored in any-hit shaders (imagining if we only hit translucent object), miss shaders will be invoked. In other cases, a closest-hit shader will be invoked because one ray is accepted in an any-hit shader or any opaque geometries are hit, where *v* will not be updated. Developers will need to set *v* to 0 in those cases. Optionally, developers can consider overriding the hit information if that's the more desired behavior (Listing 11-5).

Any-hit shaders are invoked one after another for all potential hits during the traversal for non-opaque geometries, although the specification does not guarantee that those are invoked only once for potential hits between the ray origin and the max distance, which provides opportunities to optimize the traversal. As a result, this setup may fail for some edge cases. Although `D3D12_RAYTRACING_GEOMETRY_FLAG_NO_DUPLICATE_ANYHIT_INVOCATION` is available to achieve the correct result, we chose not to do so to keep better traversal performance in general. When it goes wrong, it will cause the result to look darker, but so far we have not noticed any of those cases.

By extending this setup, we can also implement stochastic opacity. To keep this section reasonably concise, we will skip explaining the algorithm of stochastic opacity [2] and assume the reader understands how to implement it. The pipeline can render the visibility buffer with one ray and roughly one invocation of any-hit shader per surface. To render a 16-sample visibility buffer, we first cast one ray with 16 `hitT`. The any-hit shaders first check a

Listing 11-5. *Visibility ray setup: closest-hit shader and miss shader.*

```
[shader("closesthit")]
void chs(Payload payload)
{
    // If we want to get the first opaque closest hit
    payload.hitT = RayCurrent();
    payload.accumulatedVisibility = 0;
}

[shader("miss")]
void miss(Payload payload)
{
    if(payload.hitT == 0) {
        payload.hitT = 1.f / 0.f; // Set to INF.
    }
}
```

random number against the opacity value 16 times to decide whether to consider the candidate a hit for each individual entry and then update the corresponding `hitT` and call `IgnoreHit()`. Hit distances are then stored in a buffer.

Listing 11-6 shows how to compute the final color with single ray invocation of 16 samples. After tracing the ray, what's left is to sum up all 16 samples and apply alpha correction, as in Listing 11-7.

Listing 11-6. *The 16-sample stochastic opacity ray setup.*

```
struct VisPayload
{
    float hitT[16];
};
void ahs(VisPayload payload)
{
    // geometryData is constant data that bound to the shader with
    // geometry instance data.
    // Each any-hit shader includes the corresponding material shader logic.
    float opacity = materialOpacity(geometryData);
    for(int i = 0; i < 16; i++) {
        // Pass i as additional seed.
        bool opaque = opacity > random01(i);
        if(opaque) {
            if(payload.hitT[i] == 0) {
                payload.hitT[i] = RayCurrent();
            } else {
                payload.hitT[i] = min(payload.hitT[i], RayCurrent());
            }
        }
    }
    IgnoreHit();
}
```

Listing 11-7. *The 16-sample stochastic opacity shading pass.*

```
struct ShadePayload
{
    float3 finalColor[16];
    float  hitT[16]
}
void ahs(ShadePayload payload)
{
    // geometryData is constant data that bound to the shader with
    // geometry instance data.
    // Each any-hit shader includes the corresponding material shader logic.
    float opacity = materialOpacity(geometryData);
    float3 color = materialColor(geometryData);
    // Fetch 16 hitT and compare against the current T.
    float visibility = sampleVisibility(DispatchRayIndex().xy, RayCurrent())
        ;
    for(int i = 0; i < 16; i++) {
        // Pass i as additional seed.
        bool opaque = opacity > random01(i);
        if(opaque) {
            if(payload.hitT[i] == 0) {
                payload.hitT[i] = RayCurrent();
                payload.finalColor[i] = visibility * color * opacity;
            } else {
                payload.finalColor[i] = payload.hitT[i] < RayCurrent() ?
                    payload.finalColor[i] : visibility * color * opacity;
                payload.hitT[i] = min(payload.hitT[i], RayCurrent());
            }
        }
    }
    IgnoreHit();
}
```

11.6 CONCLUSION

Have we solved the translucency problem in real-time rendering with ray tracing? Absolutely not, it is still an open problem. Besides what we mentioned so far, a bigger challenge is to handle volumetric scattering. The fact that this chapter did not introduce a uniform approach is simply because it does not exist for real time, so we have to make different assumptions and estimations to render what we can afford.

The biggest gain in terms of introducing ray tracing is adding one extra tool into developers' toolboxes, which offers developers easy access to offscreen geometry information, required for many advanced effects. Although it's definitely overstated that the introduction of hardware-accelerated ray tracing magically solves everything, we are able to shift away part of the focus from tedious engine-side infrastructure work so that some offscreen information is available for the rendering pipeline—for example, different shadow map

techniques, reflection captures, etc. We can expect that more sophisticated usage of ray tracing pipelines will also help hardware architectures evolve to help with the challenge that we are facing.

ACKNOWLEDGMENTS

The author wrote this chapter to be practical and will consider it successful if you find it inspiring and helpful. The author wants to thank the Omniverse RTX renderer team at NVIDIA, a technical team with both real-time and offline rendering professionals. Many of the techniques would not have become reality without the great discussion within the team. The author also would like to thank his friends Alan Wolfe, Blagovest Taskov, Daqi Lin, Eugene D'Eon, and Tiantian Xie for reviewing the chapter, as well as the section editor Per Christensen for editing the chapter. All the render results are produced in the Omniverse RTX renderer with either real-time mode (for implementation results) or path tracing mode (for references).

REFERENCES

[1] Akenine-Möller, T., Haines, E., and Hoffman, N. *Real-Time Rendering*. A K Peters/CRC Press, 4th edition, 2018.

[2] Enderton, E., Sintorn, E., Shirley, P., and Luebke, D. Stochastic transparency. *IEEE Transactions on Visualization and Computer Graphics*, 17(08):1036–1047, 2011. DOI: 10.1109/TVCG.2010.123.

[3] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 3rd edition, 2016.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 12

MOTION BLUR CORNER CASES

Christopher Kulla[1,2] and Thiago Ize[3]
[1]Epic Games
[2]Sony Imageworks
[3]Autodesk

ABSTRACT

This chapter describes efficient ways to handle some corner cases in motion-blurred acceleration structures for ray tracing. We discuss the handling of geometries sampled at different time intervals, how to efficiently bound the movement of geometries undergoing linear transformation and linear deformation at the same time, and a heuristic for bounding volume hierarchy (BVH) construction that helps the surface area heuristic adapt to the presence of incoherent motion.

12.1 INTRODUCTION

Motion bounding volume hierarchy (MBVH) acceleration structures [2] are a cornerstone of modern production ray tracers. The basic concept is best described by considering simple linear motion. The single bounding box of a traditional BVH is replaced with a pair of bounding boxes bounding the endpoints of the motion. At render time, each ray's time is used to interpolate a bounding box for that specific intersection test (Figure 12-1). Modern ray tracing hardware, such as NVIDIA's Ampere architecture, now even includes some hardware accelerated motion blur.

What is less commonly discussed are the issues related to some practical problems surrounding multi-segment motion blur and, in particular, odd combinations of various motion key counts among the objects of a scene. For instance, one early paper [4] suggested simply using an MBVH for bottom-level acceleration structures, while ignoring it for the top-level acceleration structure (which may combine objects of various key counts). This can have disastrous effects on performance as rapid motion can cause very large bounding boxes with substantial overlap.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_12

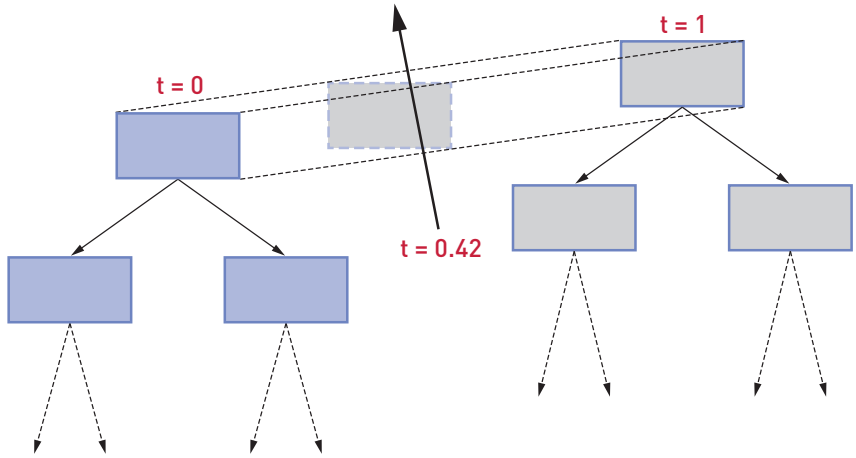


Figure 12-1. *An MBVH uses a pair of axis-aligned bounding boxes at each level in the tree that can be linearly interpolated according to a ray's time to obtain tight bounds around moving primitives.*

The classic motion spatial BVH (MSBVH) paper [3] assumes all motion segments are powers of two such that the timeline is evenly divided and some bounding boxes can be shared. The spatiotemporal BVH (STBVH) paper [13] relaxes this assumption by introducing a new kind of split in the tree along the time domain as well as extending the construction algorithm to have a time-aware surface area heuristic for detecting when such splits would be advantageous. We revisit this topic and in particular show how splitting the time interval upfront leads to an overall simpler implementation (Section 12.2) that may be advantageous for hardware implementations.

An important implementation detail is the efficient bounding of the combination of linear motion with linear transformation. Having surveyed some popular open source implementations for this task, we found that it was either ignored or handled with overly general approaches. We detail a particularly simple implementation in Section 12.3.

Finally, we show how incoherent motion can have a substantial impact on performance. We present a simple build time heuristic in Section 12.4 for MBVHs that significantly improves performance in these cases.

These methods have been successfully used in the respective Sony Pictures Imageworks and Autodesk Arnold renderers.

12.2 DEALING WITH VARYING MOTION SAMPLE COUNTS

12.2.1 MOTIVATION

Film production pipelines frequently assemble scenes authored by large teams of artists working across multiple departments. This multitude of sources can occasionally cause the composed scenes to contain objects sampled at different points in time. Moreover, it is common for a small subset of objects in the scene to require more motion keys than others. For instance, spinning wheels or propellers may require a high number of transform samples, while character motion or fluid simulations may only require linear motion (two samples).

Rendering APIs and file formats have long recognized the need for this flexibility, starting with the RenderMan Interface [10], which allowed specification of arbitrary time samples through the `MotionBegin` and `MotionEnd` calls. Today, both the Alembic and USD file formats have provisions for storing time-sampled data at arbitrary granularities.

Modern ray tracing APIs, such as Embree [12] and OptiX [8], support describing geometry starting and ending at arbitrary time values. Both impose the restriction that the samples be uniformly spaced. In practice this restriction is sensible as one otherwise needs to deal with nonuniform acceleration between different objects, which precludes the approach described next.

12.2.2 TIME SAMPLE UNIFORMIZATION

Dealing with time samples on a single piece of geometry is simple as one only needs to search for the index of the motion samples that surround the current ray time.

The more complex case arises when merging objects with different start and end times into a single structure. What we propose here is to take the union of all time samples, as shown in Figure 12-2, and decompose the timeline into non-overlapping segments. If the start and end times or sample counts do not align across objects, this will produce a nonuniform spacing of time values in the general case. However, a simple search can still be employed to find the two nearest time values for a ray with arbitrary time.

Before building the MBVH structures, we take the union of all time samples, clipped against the current camera shutter. For each consecutive pair of time

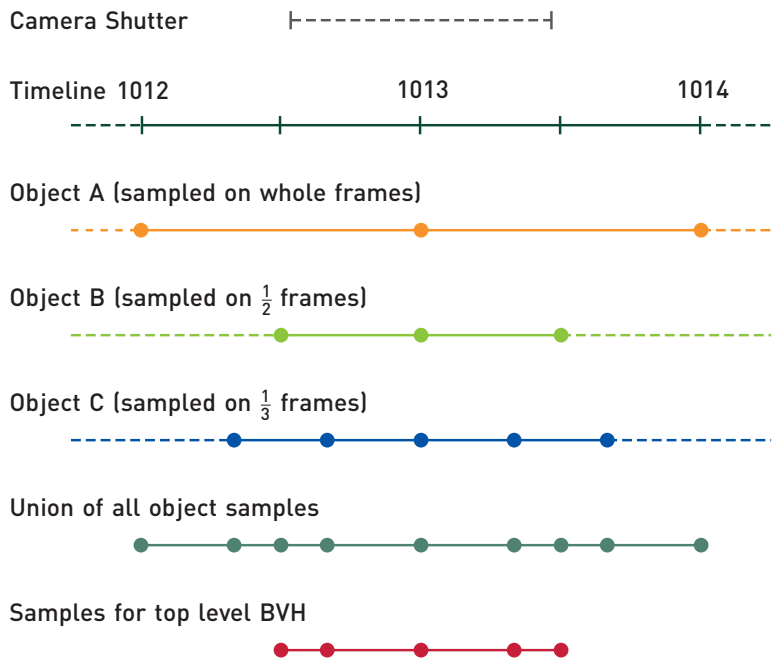


Figure 12-2. *Animation software typically represents time in frame units. Geometry may be sampled from these frames, or between frames for greater precision. When a production renderer ingests a scene, it is possible for different geometric elements to have been sampled at different rates, which yields nonuniform sample spacing for the top-level acceleration structure.*

samples, we can build an ordinary MBVH segment. In the example of Figure 12-2, there are five time samples required for the top-level BVH, or a total of four MBVH trees. When preparing the bounding boxes for each BVH, we simply ask each object for its bounding box at the specified time. In fact, many production renderers allow not only the object deformation to be time sampled, but also the object's transformation matrix. Therefore, the union of all samples should consider both together. When asking for the position of an object at an instant in time, both the interpolated object-space bounds and the interpolated transform must be provided.

The fact that the keys are not uniformly spaced is handled during traversal by a simple search to find the enclosing interval and directing the ray to the appropriate tree. When traversal of the top-level tree reaches an object, the original time is used to index the object-level BVH, which will be sampled at that object's own frequency.

For simplicity of exposition we described this process assuming that each time segment gets its own BVH tree. A more sophisticated strategy is to use a spatiotemporal BVH [13], which uses a single tree for all samples. The latter structure has the potential of being more memory efficient, but needs more careful tuning to ensure that rapid changes of motion are handled efficiently. Another drawback of the STBVH is that the handling of nonuniform time spacing will be propagated into the traversal logic. Using an array of MBVHs keeps the handling of nonuniform sample spacing outside the core traversal logic, which is appealing for implementation simplicity.

12.2.3 MOTION INTERVAL PRUNING

When following the scheme outlined in the previous section, it is possible for some time intervals of the top-level acceleration structure to be unreachable in practice. This can already be seen in the difference between the union of all object samples and the samples required for the top-level BVH in Figure 12-2. We give another concrete example here to highlight some shortcuts an implementation should take when possible.

Let us consider a simple but common scenario that mixes data from two sources: a character exported from animation software on integer frames and a particle simulation using velocity and acceleration vectors. The latter is common as particle counts typically change from frame to frame, making direct interpolation difficult. In this case, the renderer (or the application feeding the renderer) will synthesize motion samples from the velocity and acceleration vectors on the fly. The synthesis should take advantage of the currently configured camera shutter to maximize the usefulness of the motion samples. To make this concrete, let us suppose that we are exporting motion blur for frame 110 of an animation. The shutter interval is set to [109.4, 110]. The character's motion samples will correspond to {109.0, 110.0}. However, the particle data will be synthesized between shutter open and close. Let us assume that we want four motion samples. This give us particle data on {109.4, 109.6, 109.8, 110.0}. The final union of all samples in the scene is {109.0, 109.4, 109.6, 109.8, 110.0}. Naively building a MBVH per segment would lead to four structures, but due to the shutter, the first will never be accessed. It is therefore valid to prune this interval from the set. In more general configurations with additional combinations of sample counts and centered shutter intervals, this pruning can lead to significant savings.

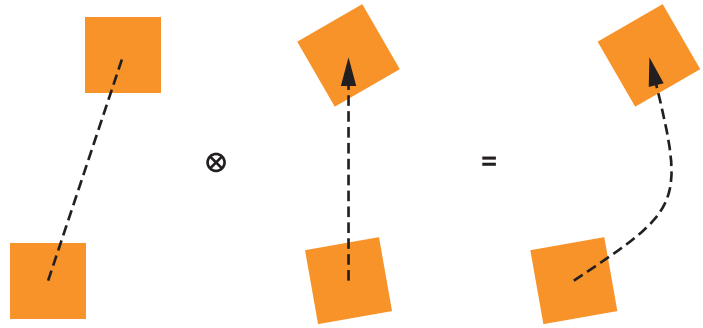


Figure 12-3. *The combination of linear deformation and linear matrix interpolation can produce a quadratic motion path.*

12.3 COMBINING TRANSFORMATION AND DEFORMATION MOTION

Most MBVH implementations are specialized to linear motion. However, when combining both transformation and deformation motion blur, it is possible for the motion path to become curved, as shown in Figure 12-3. Here, we work through the example of a linearly interpolated matrix that transforms a linearly interpolated point. This leads to a quadratic motion path that can nonetheless be bounded efficiently as linear motion.

We start from a linearly interpolating point and matrix:

$$P(t) = (1 - t)P_0 + tP_1, \tag{12.1}$$

$$\mathbf{M}(t) = (1 - t)\mathbf{M}_0 + t\mathbf{M}_1. \tag{12.2}$$

Though interpolating matrices in this way is somewhat naive, it has the advantage of inducing a purely linear motion regardless of the transform. More expressive matrix decomposition and interpolation techniques will create curved motion blur, which requires a more complex analysis. We refer the reader to the discussion in *Physically Based Rendering* on animating transformations [9, Chapter 2.9] for a good introduction to these higher-order techniques. One observation worth making is that matrix decomposition followed by interpolation of the constituents does not generally lead to the same result as interpolating a hierarchy of transform matrices individually. This leads some rendering software to need to offer additional controls for the user to hint at the pivot location for rotations, for instance.

Composing Equations 12.1 and 12.2 gives the motion of the point in world space (where the instance-level acceleration structure must be constructed).

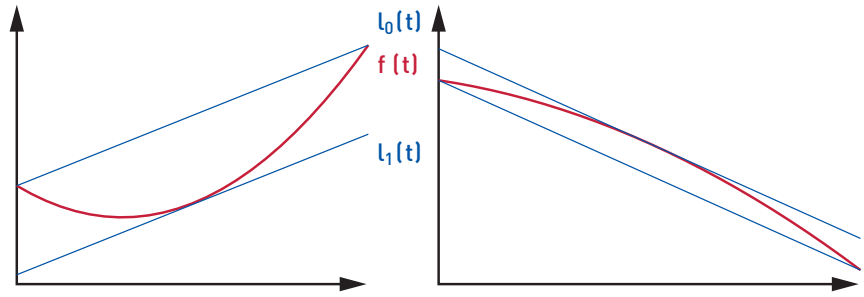


Figure 12-4. *Parabolic arcs can always be bounded by two parallel lines (Equations 12.5 and 12.6). Here, the horizontal axis represents time $t \in [0, 1]$ and the vertical axis represents one of the three spatial coordinates. The two parallel lines can be interpreted as the interpolation of 1D bounding intervals along t. Regardless of what shape the parabola takes, one line will interpolate the endpoints, while the other will be tangent to the curve.*

Expanding the expression shows that it is a quadratic arc:

$$\begin{aligned} \mathbf{M}(t)P(t) &= \left((1-t)\mathbf{M}_0 + t\mathbf{M}_1\right)\left((1-t)P_0 + tP_1\right) \\ &= A + tB + t^2C, \\ \text{where } A &= \mathbf{M}_0 P_0, \\ B &= \mathbf{M}_0\left(P_1 - 2P_0\right) + \mathbf{M}_1 P_0, \\ C &= \left(\mathbf{M}_0 - \mathbf{M}_1\right)\left(P_0 - P_1\right). \end{aligned} \tag{12.3}$$

Knowing that the point after transformation traces out a parabolic arc, we now wish to find a way to bound this motion with tight linear bounds. A quadratic polynomial over [0, 1] can be tightly bounded by two parallel lines (see Figure 12-4) as follows:

$$f(t) = a + tb + t^2 c, \tag{12.4}$$

$$l_0(t) = a + t(b + c), \tag{12.5}$$

$$l_1(t) = a + t(b + c) - \frac{1}{4}c. \tag{12.6}$$

Combining the two derivations in Equations 12.5 and 12.6 and noticing that $A + B + C = \mathbf{M}_1 P_1$, we obtain the following bounding lines for the arc $\mathbf{M}(t)P(t)$:

$$L_0(t) = (1-t)\mathbf{M}_0 P_0 + t\mathbf{M}_1 P_1, \tag{12.7}$$

$$L_1(t) = L_0(t) - \Delta, \tag{12.8}$$

$$\Delta = \frac{1}{4}(\mathbf{M}_0 - \mathbf{M}_1)(P_0 - P_1). \tag{12.9}$$

Listing 12-1. *Transforming interpolating bounds by interpolated matrices.*

```
void transformMovingAABB(
        const AABB objBox[2],
        const mat4 m[2],
        AABB worldBox[2])
{
    // Precompute difference matrix (3x3 is sufficient).
    const mat3 diff = 0.25f * (mat3(m[0]) - mat3(m[1]));
    worldBox[0].reset();
    worldBox[1].reset();
    // Loop over axis-aligned bounding box corners.
    for (int i = 0; i < 8; i++) {
        vec3 p0 = objBox[0].corner(i);
        vec3 p1 = objBox[1].corner(i);

        // Transform each endpoint.
        vec3 m0p0 = m[0] * p0;
        vec3 m1p1 = m[1] * p1;

        // Tweak for quadratic arcs.
        vec3 delta = diff * (p0 - p1);

        // Grow output boxes.
        worldBox[0].expand(m0p0);
        worldBox[0].expand(m0p0 - delta);
        worldBox[1].expand(m1p1);
        worldBox[1].expand(m1p1 - delta);
    }
}
```

Looking at the final result, Equation 12.7 is simply the linear interpolation of the endpoints. The offset Δ in Equation 12.9 is simply the offset needed to ensure that we capture the quadratic arc. We can also see that the quadratic arc only occurs when $\Delta \neq 0$, which happens when both $\mathbf{M}_0 \neq \mathbf{M}_1$ and $P_0 \neq P_1$. In fact, only the upper 3×3 portion of the matrices is relevant here because $P_0 - P_1$ is a vector. Therefore, only the scale and rotation parts of the transform can induce this nonlinear behavior.

We conclude this section by providing a simple implementation of the logic presented in the context of top-level BVH construction. Given a pair of matrices $\mathbf{M}$ and object bounding boxes B, Listing 12-1 returns a pair of bounding boxes in world space that linearly bound the curved path swept by the transformed boxes.

12.4 INCOHERENT MOTION

The last corner case we will discuss is incoherent motion, loosely explained as occurring when primitives that are near one another for one key, end up far

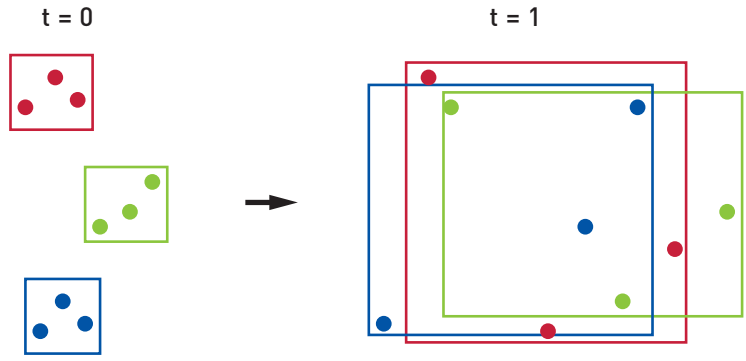


Figure 12-5. *A good partitioning of primitives at one instant in time can degrade arbitrarily badly under rapid motion, which can severely impact traversal performance if the BVH build process only considers one instant in time.*

apart in the next key and, equivalently, far-apart primitives end up spatially near. This is a well-known problem in non-motion blurred animations where BVH refitting between keys can produce a slow BVH and is often solved by reverting to full rebuilds [7], though other approaches, such as partial rebuilds [6], do exist. This approach of changing the BVH tree topology does not map well to MBVHs where our BVH interpolation is essentially an on-demand refit interpolated between two keys of identical topology.

When evaluating the surface area heuristic (SAH) during MBVH construction, it is common to simplify the problem by only looking at one instant in time. As with refitting of non-motion BVHs, one naive approach that is usually effective is to use the bound information from the first time sample for SAH calculations and carry over the topology decisions to the second key. Several papers [5, 13] recommend using the average of both bounds instead for SAH calculations.

As shown in Figure 12-5, such simple strategies can occasionally fail and cause the resulting tree to be quite poor for rays that are far from the time at which the SAH was evaluated. One possibility to improve such situations, borrowed from the animated but non-motion BVH literature, is to extend the SAH by considering both endpoints of the linear motion [11, 1]. However, this extra computation and data can slow down build performance. Considering that MBVHs tend to capture less than 40 ms of motion (when rendering for a 24 FPS film), the vast majority of motion will be coherent, which means that

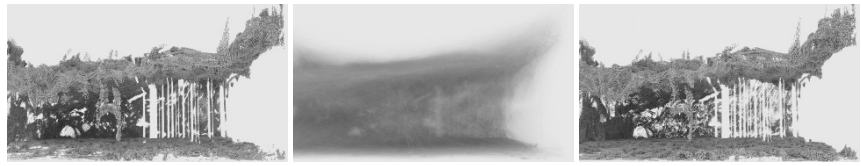


Figure 12-6. *Though key 0 (left) and key 1 (right) look similar, the interpolation between points (center) is done between random pairs of points, causing extremely incoherent motion. This error in data submission to the renderer was hard to detect because the scene could not be rendered in a reasonable time without our heuristic.*

this extra time spent in BVH build computation will in the common case likely not justify the resulting minor render time speedup, especially when time to first pixel also matters. Likewise, more complex structures, such as the STBVH that also uses temporal splits, introduce extra complexity that is not needed in most situations.

Our aim is to improve the rare case of incoherent motion by tweaking the standard hierarchy construction without imposing any new requirements on the traversal logic. We do not want to negatively affect the performance in the common case where the motion is mostly coherent among the primitives. We found that simply measuring the SAH as usual but switching the key between build levels gave us most of the quality improvements to the tree without affecting the performance of the build process or render time of coherent motion. If the key being evaluated is a leaf, we switch to another key in case it could be further subdivided. Note that this works best when the number of keys is less than the tree depth, so that all the keys can influence the build multiple times. This limitation is not too onerous because animators and physics simulations usually have coherent movement within this 40 ms window.

Despite the simplicity of our proposal, the achieved speedups can be dramatic. The impetus for this optimization is a particle motion seafoam render, shown in Figure 12-6 with simple shading, whose render time with production shaders and sample rates was originally so long that artists were unable to wait for it to finish rendering. After our optimization, artists were able to render it and then see that the incoherent particle motion was due to a bug in the indexing of particles in the code sending data to the renderer. In Table 12-1 we present the render times using just 1 spp and simple shading, which shows us achieving orders of magnitude speedups. Being robust to

	Traditional			Alternating Key		
Shutter Interval	0	[0, 1]	1	0	[0, 1]	1
Seafoam CPU	0.1 s	221 s	1137 s	2.7 s	1.3 s	1.1s
Seafoam RTX 8000	0.2 s	111 s	443 s	—	—	—
Seafoam RTX 3080	0.1 s	27 s	101 s	—	—	—
Storks CPU	13 s	101 s	185 s	14 s	25 s	19s
Storks RTX 8000	14 s	88 s	185 s	—	—	—
Storks RTX 3080	15 s	22 s	28 s	—	—	—

Table 12-1. *Render time when the shutter interval is fixed on key 0, uniformly interpolated between key 0 and key 1, or on key 1. Seafoam was rendered with 1 spp, while storks used adaptive sampling so all images had equal noise levels; both used simple shading. The traditional build method on CPU uses key 0 to build the BVH, while GPU used the standard OptiX 6.8 Trbvh. OptiX, as of this publication, does not currently have an alternating key build, so no results are reported.*

such degenerate cases is extremely important for the perceived robustness of the renderer. Note that though our heuristic for this extreme example results in slower performance when rendering only at key 0, it has consistent performance for a random time value. In contrast, heuristics that only consider a single instant in time only achieve high performance near that instant and have dramatically slower performance everywhere else.

Our second example comes from an actual movie production. In this scene, artists reported a single frame being much slower to render than the rest of the animation. Upon closer inspection, the character's hair undergoes a rapid change within that one frame (see Figure 12-7). Again, without the presented heuristic, the frame would render too slowly to even inspect if the image was correct or not.

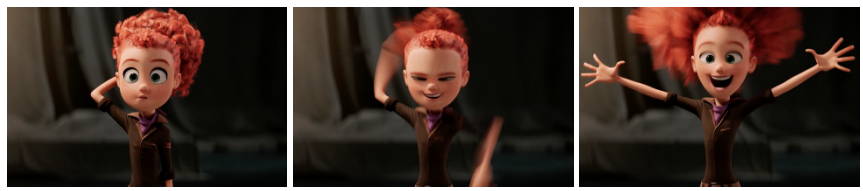


Figure 12-7. *The character's hair, modeled as B-spline curves, undergoes a rapid, sub-frame transition over the course of just two motion keys in this example from the movie Storks. (Image courtesy Warner Animation Group. ©2016 Warner Bros. Ent. Inc. All rights reserved.)*

12.5 CONCLUSION

The techniques in this chapter cover some of the less-discussed aspects of motion blur acceleration structures. As ray tracing APIs and hardware implementations take over the low-level details of ray intersection in the coming years, we expect these types of discussions of corner cases to be relevant to establishing best practices across implementations.

REFERENCES

[1] Bittner, J. and Meister, D. T-SAH: Animation optimized bounding volume hierarchies. *Computer Graphics Forum*, 34:527–536, 2015. DOI: 10.1111/cgf.12581.

[2] Christensen, P. H., Fong, J., Laur, D. M., and Batali, D. Ray tracing for the movie 'Cars'. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, pages 1–6, 2006. DOI: 10/bmph7q.

[3] Grünschloß, L., Stich, M., Nawaz, S., and Keller, A. MSBVH: An efficient acceleration data structure for ray traced motion blur. In *Proceedings of High Performance Graphics*, pages 65–70, 2011. DOI: 10.1145/2018323.2018334.

[4] Hanika, J., Keller, A., and Lensch, H. P. A. Two-level ray tracing with reordering for highly complex scenes. In *Proceedings of Graphics Interface*, pages 145–152, 2010.

[5] Hou, Q., Qin, H., Li, W., Guo, B., and Zhou, K. Micropolygon ray tracing with defocus and motion blur. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 29(4):64:1–64:10, 2010. DOI: 10.1145/1833349.1778801.

[6] Kopta, D., Ize, T., Spjut, J., Brunvand, E., Davis, A., and Kensler, A. Fast, effective BVH updates for animated scenes. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pages 197–204, 2012.

[7] Lauterbach, C., Yoon, S.-E., Tuft, D., and Manocha, D. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, pages 39–46, 2006. DOI: 10.1109/RT.2006.280213.

[8] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 29(4):66:1–66:13, July 2010. DOI: 10/frf4mq.

[9] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 3rd edition, 2016.

[10] Upstill, S. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[11] Wald, I., Boulos, S., and Shirley, P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics*, 26(1):1–18, 2007. DOI: 10.1145/1189762.1206075.

[12] Wald, I., Woop, S., Benthin, C., Johnson, G. S., and Ernst, M. Embree: A kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 33(4):143:1–143:8, 2014. DOI: 10/gfzwck.

[13] Woop, S., Áfra, A. T., and Benthin, C. STBVH: A spatial-temporal BVH for efficient multi-segment motion blur. In *Proceedings of High Performance Graphics*, 8:1–8:8, 2017. DOI: 10.1145/3105762.3105779.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 13

FAST SPECTRAL UPSAMPLING OF VOLUME ATTENUATION COEFFICIENTS

Johannes Jendersie
NVIDIA

ABSTRACT

Most of the input data for rendering is authored in tristimulus colors. In a spectral renderer these are usually converted into spectra that have more than three degrees of freedom while maintaining the authored appearance under a white illuminant. For volumes there are two additional problems: (1) the nonlinear color shift over the distance traveled through the volume and (2) the large amount of data or procedural data. Problem 1 makes it difficult to preserve the appearance, whereas problem 2 makes it infeasible to precompute the spectral data. This chapter shows that using box-like spectra produces the closest fit to a tristimulus render of the tristimulus data. We propose to optimize two threshold wavelengths depending on the input color space that are then used to convert the volume coefficients on the fly.

13.1 INTRODUCTION

Our goal is to maximize the similarity of a spectral volume rendering compared to the rendering in tristimulus color space. Thereby, a volume is parameterized by coefficients $\sigma_t = \sigma_s + \sigma_a$ in $[0, \infty]$, where σ_t is the attenuation (also called extinction), σ_s is the scattering, and σ_a the absorption coefficient.

The amount of light of wavelength λ reaching a specific distance ℓ along a ray is called transmittance T. Without loss of generality, let us consider homogeneous media, where the transmittance is described by the Beer–Lambert law:

$$T(\ell, \lambda) = e^{-\ell \cdot \sigma_t(\lambda)}. \tag{13.1}$$

Clearly, the chromaticity changes over distance, if σ_t is different for different wavelengths due to the nonlinearity of the transmittance. When rendering in RGB color space, the attenuation coefficient is a triple $\boldsymbol{\sigma_t} = (\sigma_r, \sigma_g, \sigma_b)$. An

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_13

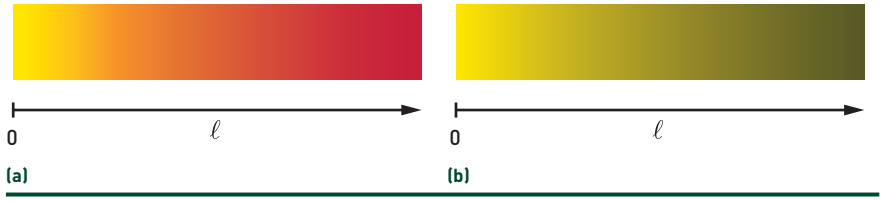


Figure 13-1. *Volume attenuation $\boldsymbol{\sigma}_t = (1, 10, \infty)$ of (a) yellow light and (b) with spectral wavelength $\lambda = 580$ nm ($\sigma_t(\lambda) = 5.5$). In (a) red light is absorbed the least, producing a color shift.*

example is demonstrated in Figure 13-1a. Figure 13-1b shows a spectral rendering example for a monochromatic yellow illuminant. The perceived color at the very left is the same as in Figure 13-1a, but there is no color shift to the right.

Further, we desire linearity $m(\boldsymbol{\sigma}_t) = m(\boldsymbol{\sigma}_s) + m(\boldsymbol{\sigma}_a)$ of the solution, where $m(x)$ is the mapping from RGB to spectral colors. If this would not be the case, we could still convert only two of the values and derive the third one, but this would make the behavior less predictable when switching between parameterizations.

Another problem when rendering inhomogeneous volumes is the vast amount of data. When converting tristimulus values into spectra, those usually require more memory for their representation. Another possibility is the existence of procedural data for which we want to compute the conversion on the fly. Therefore, the solution should be able to directly convert colors fast, without an expensive optimization.

13.1.1 KNOWN SOLUTIONS

The problem of upsampling a tristimulus color into a spectrum is well researched and has received increased attention in the last few years [5, 2]. Most of the approaches aim to create smooth spectra and to enforce energy preservation of reflectance values, for example by keeping the spectral values in $[0, 1]$. In contrast, our target is to convert attenuation coefficients in $[0, \infty]$.

In the book *Physically Based Rendering* [6], Chapter 5 gives a practical introduction into spectral rendering, where for the conversion of RGB to spectral colors, the method from Smits [8] is used. It converts RGB colors fast by creating a linear combination of seven precomputed discrete spectra. The focus of the method is to produce smooth spectra that will exhibit strong color shifts in volume renderings.

The approach of Otsu et al. [5] relies on measured spectra that are converted into base functions for runtime conversion. Selecting and evaluating the base functions is relatively costly if done for each fetch in an inhomogeneous volume renderer, and it would require measured spectra of volume attenuation coefficients. However, their approach would produce more realistic spectra than our method. Jakob and Hanika [2] also proposed an approach that would be feasible with respect to data amount and evaluation speed using a nonlinear parametric function. However, it is applicable to neither procedural data nor the unbounded attenuation coefficients.

The approach most similar to our solution is that of MacAdam [3], who showed that the (inverted) box function maximizes the brightness of a target saturation. Though we are using box-shaped spectra, too, we apply the conversion to attenuation coefficients. Instead of optimizing the spectrum for a single target color, we optimize thresholds to be used over all colors to achieve a very fast conversion algorithm.

13.2 PROPOSED SOLUTION

Equation 13.1 already introduced how light of a single wavelength or channel is attenuated. Let us now draw a connection between the tristimulus and the spectral models. Formally, let $I(\lambda)$ be the spectral illuminant and $c(\lambda)$ any response curve to convert a spectrum into one of the tristimulus values. This can be any curve for XYZ or RGB space directly as defined by CIE 1931 [7, 4, 1]. Then, the light reaching a specific depth in the volume converted into a tristimulus color channel c is

$$L_c(\ell) = \int c(\lambda) \cdot I(\lambda) \cdot e^{-\ell \cdot \sigma_t(\lambda)} d\lambda. \tag{13.2}$$

Alternatively, we have

$$L'_c(\ell) = I_c \cdot e^{-\ell \cdot \sigma_t(c)} \tag{13.3}$$

to compute the same value in the respective color channel directly. Setting L_c and L'_c to be equal, we get

$$I_c \cdot e^{-\ell \cdot \sigma_t(c)} = \int c(\lambda) \cdot I(\lambda) \cdot e^{-\ell \cdot \sigma_t(\lambda)} d\lambda, \tag{13.4}$$

which can only become true if $\sigma_t(c) = \sigma_t(\lambda)$ is a constant wherever $c(\lambda) \cdot I(\lambda) \neq 0$ and if $I_c = \int c(\lambda) I(\lambda) d\lambda$. The second condition means that I_c must be the channel response of the used illuminant. It is reasonable to use $I_c = 1$

together with the white point of the target color space, which for example would be D65 for sRGB color space.

As suggested by Equation 13.4, we have to set $\sigma_t(\lambda) = \sigma_t(c)$ wherever $c(\lambda) > 0$. However, we have three color channels with different attenuation coefficients σ_r, σ_g, and σ_b that overlap each other. We found that selecting the coefficient of the color channel with the highest response value at the target wavelength works best. Effectively, this subdivides the visible spectrum into three parts. Thus, the function executed for conversion is

$$\sigma_t(\lambda) = \begin{cases} \sigma_b & \text{if } \lambda < \lambda_0, \\ \sigma_g & \text{if } \lambda_0 \le \lambda < \lambda_1, \\ \sigma_r & \text{otherwise,} \end{cases} \tag{13.5}$$

where the thresholds λ_0 and λ_1 are subject to a prior optimization process.

13.2.1 OPTIMIZING THRESHOLD VALUES

As a target function, we propose to minimize the deviation of the chromaticity xy of the three primaries under the color spaces' white illuminant I:

$$\begin{aligned} \underset{\lambda_0,\lambda_1}{\operatorname{argmin}} \big(& \| \mathrm{s2xy}(I \cdot \mathrm{box}_{\lambda_{\min}}^{\lambda_0}) - xy_b \|_2 \cdot \\ & + \| \mathrm{s2xy}(I \cdot \mathrm{box}_{\lambda_0}^{\lambda_1}) - xy_g \|_2 \\ & + \| \mathrm{s2xy}(I \cdot \mathrm{box}_{\lambda_1}^{\lambda_{\max}}) - xy_r \|_2 \big), \end{aligned} \quad \text{with} \quad \mathrm{box}_{x_0}^{x_1}(\lambda) = \begin{cases} 1 & \text{if } \lambda \in [x_0, x_1), \\ 0 & \text{otherwise,} \end{cases} \tag{13.6}$$

where s2xy is the conversion procedure from a spectrum to a chromaticity value (spectrum $\rightarrow$ XYZ $\rightarrow$ xy) and xy_r, xy_g, and xy_b are the defined chromaticity coordinates of the primaries (usually red, green and blue). The box functions with shared boundaries separate the spectrum into three disjoint intervals over the wavelengths. The difference to the primary makes sure to select those wavelengths that have a dominant effect on the respective color channel.

13.2.2 EXAMPLE OPTIMIZED VALUES

We optimized the values for a spectrum of $\lambda \in [380\,\text{nm}, 780\,\text{nm}]$ with 5 nm discretization steps for selected color spaces, yielding the thresholds in Table 13-1.

	XYZ	sRGB	ACES	ACEScg	Rec2020
λ_0	500	485	505	505	500
λ_1	575	595	550	570	570

Table 13-1. *Optimized thresholds for selected color spaces.*

13.3 RESULTS

The arguments in Section 13.2 show that only constant spectra will reproduce the color shift over distances ℓ within a volume. The box-shaped spectra will maximize the similarity for a tristimulus color space, because they assign the largest possible constant support region for each color channel. Hence, a smoother spectrum must deviate more. We use the smooth function to demonstrate the differences:

$$\sigma_{tI}(\lambda) = \begin{cases} \sigma_b & \text{if } \lambda < \lambda_b, \\ s\left(\sigma_b, \sigma_g, \frac{\lambda-\lambda_b}{\lambda_g-\lambda_b}\right) & \text{if } \lambda_b \le \lambda < \lambda_g, \\ s\left(\sigma_g, \sigma_r, \frac{\lambda-\lambda_g}{\lambda_r-\lambda_g}\right) & \text{if } \lambda_g \le \lambda < \lambda_r, \\ \sigma_r & \text{otherwise,} \end{cases} \tag{13.7}$$

where $s(x_0, x_1, t) = x_0 + (x_1 - x_0) \cdot (6t^5 - 15t^4 + 10t^3)$ is the smootherstep interpolation function and $\lambda_r, \lambda_g, \lambda_b$ are optimized values for the three support wavelengths of the color channels.

Figure 13-2 shows an example spectrum and renderings with both types of spectral representations. Though many parameterizations look similar, the smooth spectrum (top) fails to approximate the intent for some configurations. Figure 13-3 plots the transmittance over ℓ for different coefficients. Note that the smooth version deviates most where the red and blue channels are both different from the green one.

13.4 CONCLUSION

This chapter showed that we can use a simple box-shaped spectrum to directly select the volume attenuation coefficient for a given wavelength λ (Equation 13.5). This is both extremely fast and highly similar to the tristimulus authored data. Further, it has the advantage of being linear such that we can convert individual coefficients and sums of coefficients with the same result.

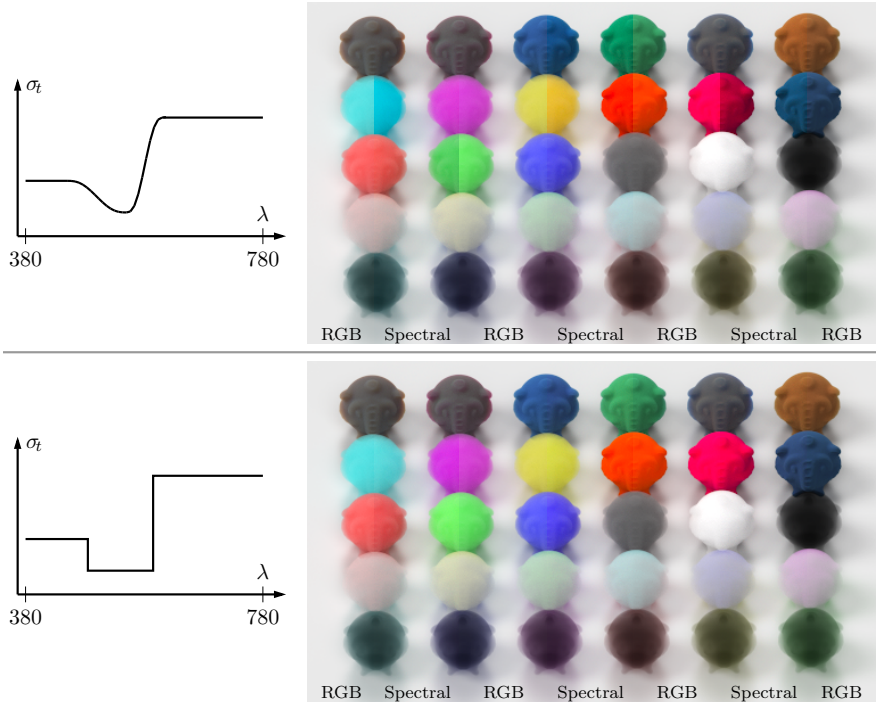


Figure 13-2. *Comparison of RGB and spectral rendering for volume coefficients given in RGB. The proposed method using box spectra (bottom) is closer to the RGB reference than the smooth spectrum (top).*

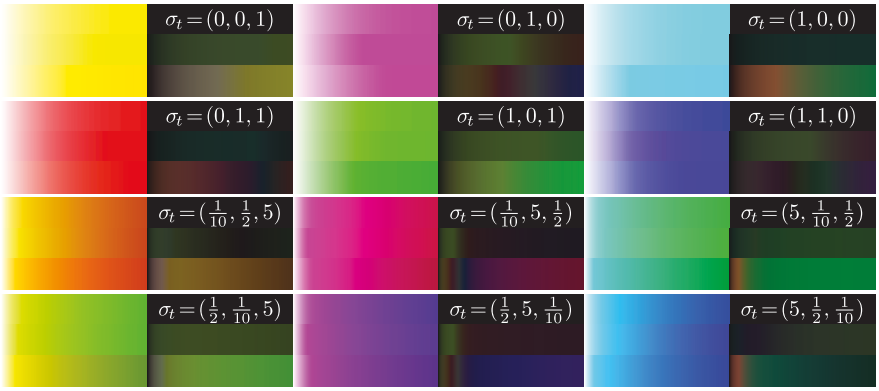


Figure 13-3. *Transmittance for a number of selected attenuation coefficients in sRGB color space (left boxes) and their difference to the RGB rendering (right boxes). Each block shows RGB, σ_t (Equation 13.5), and σ_{tI} (Equation 13.7) from top to bottom.*

REFERENCES

[1] Fairman, H. S., Brill, M. H., and Hemmendinger, H. How the CIE 1931 color-matching functions were derived from Wright-Guild data. *Color Research & Application*, 22(1):11–23, 1997. DOI: 10.1002/(SICI)1520-6378(199702)22:1<11::AID-COL4>3.0.CO;2-7.

[2] Jakob, W. and Hanika, J. A low-dimensional function space for efficient spectral upsampling. *Computer Graphics Forum*, 38(2):147–155, 2019. DOI: 10.1111/cgf.13626.

[3] MacAdam, D. L. Maximum visual efficiency of colored materials. *Journal of the Optical Society of America*, 25(11):361–367, 1935. DOI: 10.1364/JOSA.25.000361.

[4] Nimeroff, I. Field trial of the 1959 CIE supplementary standard observer proposal. *Journal of the Optical Society of America*, 54(5):696–704, 1964. DOI: 10.1364/JOSA.54.000696.

[5] Otsu, H., Yamamoto, M., and Hachisuka, T. Reproducing spectral reflectances from tristimulus colours. *Computer Graphics Forum*, 37(6):370–381, 2018. DOI: 10.1111/cgf.13332.

[6] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., 3rd edition, 2016.

[7] Smith, T. and Guild, J. The C.I.E. colorimetric standards and their use. *Transactions of the Optical Society*, 33(3):73–134, 1931. DOI: 10.1088/1475-4878/33/3/301.

[8] Smits, B. An RGB-to-spectrum conversion for reflectances. *Journal of Graphics Tools*, 4(4):11–22, 1999. DOI: 10.1080/10867651.1999.10487511.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 14

THE REFERENCE PATH TRACER

Jakub Boksansky and Adam Marrs
NVIDIA

ABSTRACT

The addition of ray tracing to a real-time renderer makes it possible to create beautiful dynamic effects—such as soft shadows, reflections, refractions, indirect illumination, and even caustics—that were previously not possible. Substantial tuning and optimization are required for these effects to run in real time without artifacts, and this process is simplified when the real-time rendering engine can progressively generate a "known-correct" reference image to compare against. In this chapter, we walk through the steps to implement a simple, but sufficiently featured, path tracer to serve as such a reference.

14.1 INTRODUCTION

With the introduction of the DirectX Raytracing (DXR) and Vulkan Ray Tracing (VKR) APIs, it is now possible to integrate ray tracing functionality into real-time rendering engines based on DirectX and Vulkan. These new APIs provide the building blocks necessary for ray tracing, including the ability to (1) quickly construct spatial acceleration structures and (2) perform fast ray intersection queries against them. Critically, the new APIs provide ray tracing operations with full access to memory resources (e.g., buffers, textures) and common graphics operations (e.g., texture sampling) already used in rasterization. This creates the opportunity to reuse existing code for geometry processing, material evaluation, and post-processing and to build a hybrid renderer where ray tracing works in tandem with rasterization.

A noteworthy advantage of a hybrid ray–raster renderer is the ability to choose how the scene is sampled. Rasterization provides efficient spatially coherent ray and surface material evaluations from a single viewpoint,[1] whereas ray tracing provides convenient incoherent ray–surface evaluation from any point in any direction. The flexibility afforded by arbitrary ray casts

[1] NVIDIA's RTX 2000 series added the ability to rasterize four arbitrary viewpoints at once, but generalized hardware multi-view rasterization is not common.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_14



Figure 14-1. *An interior scene from Evermotion's Archinteriors Vol. 48 for the Blender package [5] rendered with our reference path tracer using 125,000 paths per pixel.*

enables a hybrid real-time renderer to more naturally generate and accumulate accurate samples of a scene to construct a reference image while not being limited by the artifacts caused by discretization during rasterization (e.g., see shadow mapping).

In this chapter, we discuss how to implement a path tracer in a hybrid renderer to produce in-engine reference images for comparison with the output of new algorithms, approximations, and optimizations used in the real-time code path. We discuss tricks for handling self-intersection, importance sampling, and the evaluation of many light sources. We focus on progressively achieving "ground truth" quality and prefer simple, straightforward code over optimizations for runtime performance. Although we do not discuss optimization or the reduction of artifacts such as noise, improvements in these areas are described in other chapters of this book.

Figure 14-1 is a reference image of an interior scene from Evermotion's Archinteriors Vol. 48 for the Blender package [5]. The image is rendered progressively by our DXR-based path tracer and includes 125,000 paths per pixel with up to ten bounces per path. To help you add a reference path tracer to your renderer, the full C++ and HLSL source code of our path tracer is freely available at the book's source code website and at https://github.com/boksajak/referencePT.

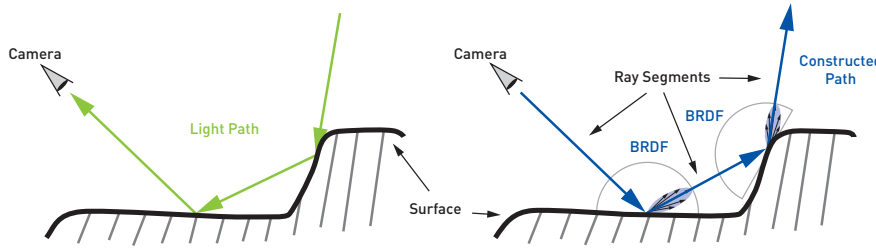


Figure 14-2. *Monte Carlo Unidirectional Path Tracing. Left: light energy bounces around the environment on its way to the camera. Right: the path tracer's model of the light transport from the left diagram. Surface material BRDFs are stochastically sampled at each point where a ray intersects a surface.*

14.2 ALGORITHM

When creating reference images, it is important to choose an algorithm that best matches the use case. A path tracer can be implemented in many ways, and that variety of algorithms is accompanied by an equivalent amount of trade-offs. An excellent overview of path tracing can be found in the book *Physically Based Rendering* [11]. Since this chapter's focus is simplicity and high image quality, we've chosen to implement a *Monte Carlo Unidirectional Path Tracer*. Let's break down what this means—starting from the end and working backward:

1. *Path Tracing* simulates how light energy moves through an environment (also known as *light transport*) by constructing "paths" between light sources and the virtual camera. A *path* is the combination of several ray segments. *Ray segments* are the connections between the camera, surfaces in the environment, and/or light sources (see Figure 14-2).

2. *Unidirectional* means paths are constructed in a single (macro) direction. Paths can start at the camera and move toward light sources or vice versa. In our implementation, paths begin exclusively at the camera. Note that this is opposite the direction that light travels in the physical world!

3. *Monte Carlo* algorithms use random sampling to approximate difficult or intractable integrals. In a path tracer, the approximated integral is the rendering equation, and tracing more paths (a) increases the accuracy of the integral approximation and (b) produces a better image. Since paths are costly to compute, we accumulate the resulting color from each path over time to construct the final image.

14.3 IMPLEMENTATION

Before implementing a reference path tracer in your engine, it is helpful to understand the basics of modern ray tracing APIs. DXR and VKR introduce new shader stages (*ray generation*, *closest-hit*, *any-hit*, and *intersection*), acceleration structure building functions, ray dispatch functions, and shader management mechanisms. Since these topics have been covered well in previous literature, we recommend Chapter 3, "Introduction to DirectX Raytracing," [19] of *Ray Tracing Gems* [6], the SIGGRAPH course of the same name [18], and Chapter 16 of this book to get up to speed. For a deeper understanding of how ray tracing works agnostic of API specifics, see Shirley's *Ray Tracing in One Weekend* series [13].

The code sample accompanying this chapter is implemented with DXR and extends the freely available IntroToDXR sample [10]. At a high level, the steps necessary to perform GPU ray tracing with the new APIs are as follows:

- At startup:
  - Initialize a DirectX device with ray tracing support.
  - Compile shaders and create the ray tracing *pipeline state object*.
  - Allocate memory for and initialize a *shader table*.
- Main loop:
  - Update *bottom-level acceleration structures* (BLAS) of geometry.
  - Update *top-level acceleration structures* (TLAS) of instances.
  - Dispatch groups of *ray generation shaders*.

With the basic execution model in place, the following sections describe the implementation of the key elements needed for a reference path tracer.

14.3.1 ACCELERATION STRUCTURE MEMORY

In-depth details regarding acceleration structure memory allocation are often omitted (rightfully so) from beginner ray tracing tutorials; however, careful acceleration structure memory management is a higher-priority topic when integrating a GPU-based path tracer into an existing rendering engine. There are two scenarios where memory is allocated for acceleration structures: (1) to store built BLAS and TLAS and (2) for use as intermediate "scratch" buffers by the API runtime during acceleration structure builds.

Memory for built acceleration structures can be managed using the same system as memory for geometry, since their lifetime is coupled with the meshes they represent. Scratch buffers, on the other hand, are temporary and may be resized or deallocated entirely once acceleration structure builds are complete. This presents an opportunity to decrease the total memory use with more adept management of the scratch buffer.

A basic scratch buffer management strategy allocates a single memory block of predefined size to use for building all acceleration structures. When the memory needed to build the current acceleration structures exceeds the predefined capacity, the memory block is increased in size. Acceleration structures typically require a small amount of memory (relative to the quantity available on current GPUs), and the maximum memory use can be predetermined in many cases with automated fly-throughs of scenes. Nevertheless, this approach's strengths *and* weaknesses stem from its simplicity. Since any part of the memory block may be in use, buffer resize and/or deallocation operations must be deferred until *all* acceleration structures are built—leading to worst-case memory requirements.

To reduce the total memory use, an alternative approach instead serializes BLAS builds (to some degree) and reuses scratch buffer memory for multiple builds. Barriers are inserted between builds to ensure that the scratch memory blocks are safe to reuse before upcoming BLAS build(s) execute. This process may occur within the scope of a single frame or across several frames. The number of BLAS builds that execute in parallel versus the maximum memory needed to build the current group of BLAS is a balancing act driven by the application's constraints and the target hardware's capabilities. Amortizing builds across multiple frames is especially useful when a scene contains a large number of meshes.

14.3.2 PRIMARY RAYS

Time to start tracing rays! We begin by implementing a ray generation shader that traces primary (camera) rays. This step is a quick way to get our first ray traced image on screen and confirm that the ray tracing pipeline is working.

To make direct comparisons with output from a rasterizer, we construct primary rays by extracting the origin and direction from the same 4×4 view and projection matrices used by the rasterizer. We compose primary ray directions using the camera's basis, the image's aspect ratio, and the camera's vertical field of view (fov). The implementation is shown in

Listing 14-1. *HLSL code to generate primary rays that match a rasterizer's output.*

```
float2 pixel = float2(DispatchRaysIndex().xy);
float2 resolution = float2(DispatchRaysDimensions().xy);

pixel = (((pixel + 0.5f) / resolution) * 2.f - 1.f);
generatePinholeCameraRay(pixel);

RayDesc generatePinholeCameraRay(float2 pixel)
{
    // Set up the ray.
    RayDesc ray;
    ray.Origin = gData.view[3].xyz;
    ray.TMin = 0.f;
    ray.TMax = FLT_MAX;

    // Extract the aspect ratio and fov from the projection matrix.
    float aspect = gData.proj[1][1] / gData.proj[0][0];
    float tanHalfFovY = 1.f / gData.proj[1][1];

    // Compute the ray direction.
    ray.Direction = normalize(
        (pixel.x * gData.view[0].xyz * tanHalfFovY * aspect) -
        (pixel.y * gData.view[1].xyz * tanHalfFovY) +
        gData.view[2].xyz);

    return ray;
}
```

Listing 14-1. Note that the aspect ratio and field of view are extracted from the projection matrix and the camera basis right (`gData.view[0].xyz`), up (`gData.view[1].xyz`), and forward (`gData.view[2].xyz`) vectors are read from the view matrix.

If the view and projection matrices are only available on the GPU as a single combined view-projection matrix, the inverse view-projection matrix (which is typically also available) can be applied to "unproject" points on screen from normalized device coordinate space. This is not recommended, however, as near and far plane settings stored in the projection matrix cause numerical precision issues when the transformation is reversed. More information on constructing primary rays in ray generation shaders can be found in Chapter 3.

To test that primary rays are working, it is useful to include visualizations of built-in DXR variables. For example, the ray hit distance to the nearest surface (`RayTCurrent()`) creates an image similar to a depth buffer. For more colorful visualizations, we output a unique color for each instance index of the

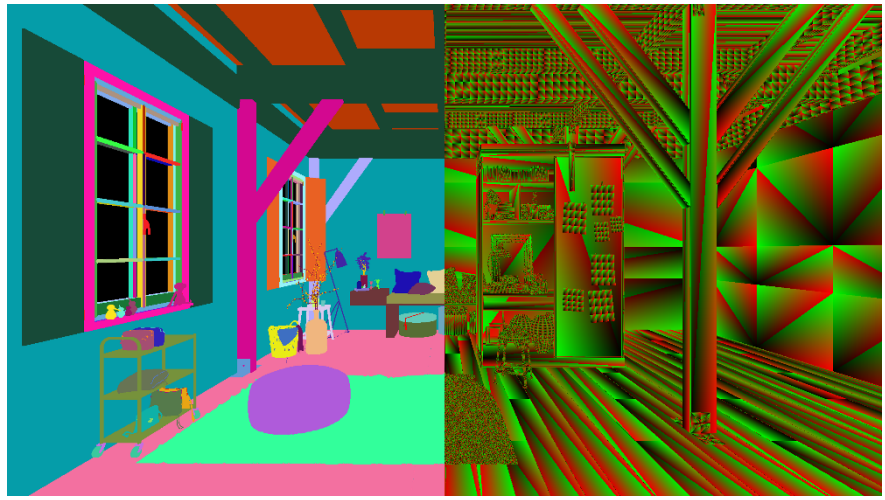


Figure 14-3. *Visualizing geometry instance indices (left) and triangle barycentrics (right) is useful when testing that primary ray tracing is working properly.*

intersected geometry (`InstanceIndex()`) and triangle barycentrics (`BuiltInTriangleIntersectionAttributes.barycentrics`). This is shown in Figure 14-3. A function for hashing an integer to a color, called `hashToColor(int)`, is available in our code sample.

It is also helpful to implement a split-view or quick-swap option to directly compare ray traced and rasterized images next to each other. Even when displaying only the albedo or hit distance, this comparison mode can reveal subtle mismatches between the ray traced and rasterized outputs. For example, geometry may be absent or flicker due to missing barriers or race conditions associated with acceleration structure builds. Another common problem is for instanced meshes to be positioned incorrectly because the DXR API expects row-major transformation matrices instead of HLSL's typical column-major packed matrices.

14.3.3 LOADING GEOMETRY AND MATERIAL PROPERTIES

Now that primary rays are traversing acceleration structures and intersecting geometry in the scene, the next step is to load the geometry and material properties of the intersected surfaces. In our code sample, we use a *bindless* approach to access resources on the GPU in both ray tracing and rasterization. This is implemented with a set of linear buffers that contain all

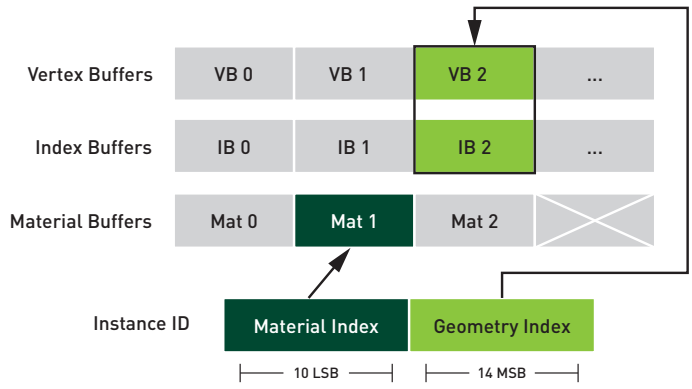


Figure 14-4. *Bindless access of geometry and material data using an encoded instance ID.*

geometry and material data used in the scene. The buffers are then marked as accessible to any invocation of any shader.

The main difference when ray tracing is that the index and vertex data of the geometry (e.g., position, texture coordinates, normals) must be explicitly loaded in the shaders and interpolated manually. To facilitate this, we encode and store the buffer indices of the geometry and material data in the 24-bit instance ID parameter of each TLAS geometry instance descriptor. Illustrated in Figure 14-4, the index into the array of buffers containing the geometry index and vertex data is packed into the 14 most significant bits of the instance ID value, and the material index is packed into the 10 least significant bits (note that this limits the number of unique geometry and material entries to 16,384 and 1,024 respectively). The encoded value is then read with `InstanceID()` in the closest (or any) hit shader, decoded, and used to load the proper material and geometry data. This encoding scheme is implemented in HLSL with two complementary functions shown in Listing 14-2.

Listing 14-2. *HLSL to encode and decode the geometry and material indices.*

```
inline uint packInstanceID(uint materialID, uint geometryID) {
    return ((geometryID & 0x3FFF) << 10) | (materialID & 0x3FF);
}

inline void unpackInstanceID(uint instanceID, out uint materialID,
                             out uint geometryID) {
    materialID = instanceID & 0x3FF;
    geometryID = (instanceID >> 10) & 0x3FFF;
}
```

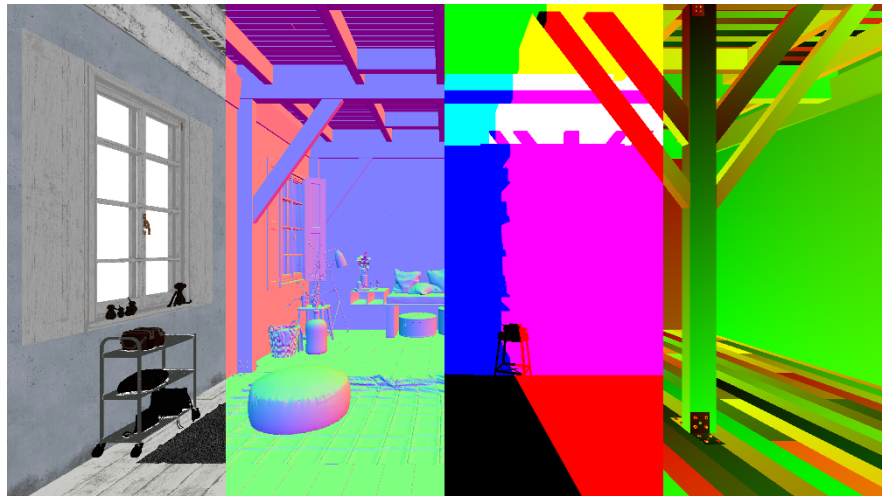


Figure 14-5. *Visualizations of various properties output by the ray generation shader. From left to right: base color, normal, world-space position, and texture coordinates.*

To confirm that the geometry and material data have been loaded and interpolated correctly, it is useful to output G-buffer–like visualizations from the ray generation shader of geometric data (e.g., world-space position, geometric normal, texture coordinates) and material properties (e.g., albedo, shading normal, roughness, etc.). These images can be directly compared with visualizations of a G-buffer generated with a rasterizer, as shown in Figure 14-5.

14.3.4 RANDOM NUMBER GENERATION

At the core of every path tracer, there is a random number generator (RNG). Random numbers are necessary to drive the sampling of materials, lights, procedurally generated textures, and much more as the path tracer simulates light transport. A high-quality RNG with a long period is an essential tool in path tracing. It ensures that each sample taken makes meaningful progress toward reducing noise and improving its approximation of the rendering equation without bias.

We rely on pseudo-random numbers, which have same statistical properties as real random numbers and are generated deterministically using one initial *seed* value. A common way to generate pseudo-random numbers on the GPU is to compute them in shader code at runtime. A fast and popular shader-based method is an xorshift function or linear congruential generator

Listing 14-3. *Initialization of the random seed for an xorshift-based RNG for the given pixel and frame number. The seed provides an initial state for the RNG and is modified each time a new random number is generated.*

```
uint jenkinsHash(uint x) {
    x += x << 10;
    x ^= x >> 6;
    x += x << 3;
    x ^= x >> 11;
    x += x << 15;
    return x;
}

uint initRNG(uint2 pixel, uint2 resolution, uint frame) {
    uint rngState = dot(pixel, uint2(1, resolution.x)) ^ jenkinsHash(frame);
    return jenkinsHash(rngState);
}
```

(LCG) seeded with a hash function [12]. First, the random seed for the RNG is established by hashing the current pixel's screen coordinates and the frame number (see Listing 14-3). This hashing ensures a good distribution of random numbers spatially across neighboring pixels and temporally across subsequent frames. We use the Jenkins's *one_at_a_time* hash [8], but other fast hash functions, such as the *Wang hash* [17], can be used as well.

Next, each new random number is generated by converting the seed into a floating-point number and hashing it again (see Listing 14-4). Notice how the `rand` function modifies the RNG's state. Since the generated number is a sequence of random bits forming an unsigned integer, we need to convert this to a floating-point number in the range [0, 1). This is achieved by using

Listing 14-4. *Generating a random number using the xorshift RNG. The `rand` function invokes the `xorshift` function to modify the RNG state in place and then converts the result to a random floating-point number.*

```
float uintToFloat(uint x) {
    return asfloat(0x3f800000 | (x >> 9)) - 1.f;
}

uint xorshift(inout uint rngState)
{
    rngState ^= rngState << 13;
    rngState ^= rngState >> 17;
    rngState ^= rngState << 5;
    return rngState;
}

float rand(inout uint rngState) {
    return uintToFloat(xorshift(rngState));
}
```

the 23 most significant bits as a mantissa of the floating-point number representation and setting the sign bits and exponent to zero. This generates numbers in the [1, 2) range, so we subtract one to shift numbers into the desired [0, 1) range.

In our code sample, we also include an implementation of the recent PCG4D generator [7]. This RNG uses an input vector of four numbers (the pixel's screen coordinates, the frame number, and the current sample number) and returns *four* random numbers. See Listing 14-5.

Listing 14-5. *Implementation of the PCG4D RNG [7]. An input vector of four numbers is transformed into four random numbers.*

```
uint4 pcg4d(uint4 v)
{
    v = v * 1664525u + 1013904223u;

    v.x += v.y * v.w;
    v.y += v.z * v.x;
    v.z += v.x * v.y;
    v.w += v.y * v.z;

    v = v ^ (v >> 16u);
    v.x += v.y * v.w;
    v.y += v.z * v.x;
    v.z += v.x * v.y;
    v.w += v.y * v.z;

    return v;
}
```

The four inputs PCG4D requires are readily available in the ray generation shader; however, they must be passed into other shader stages (e.g., closest and any-hit). For best performance, the payload that passes data between ray tracing shader stages should be kept as small as possible, so we hash the four inputs to a more compact seed value before passing it to the other shader stages. A discussion of different strategies for hashing these parameters and updating the hash for drawing subsequent samples is discussed in the recent survey by Jarzynski and Olano [7].

Both of the RNGs we've discussed generate uniform distributions (white noise) with long periods. The random numbers are generally well distributed, but there is no guarantee that similar random numbers won't appear in nearby pixels (spatially or temporally). As a result, it can be beneficial to also use *sequences* of random numbers to combat this problem. Low-discrepancy sequences, such as Halton or Sobol, are commonly used as well as blue noise. These help improve convergence speed but are typically only useful for

sampling in a few dimensions. Some sequences, such as blue noise, must be precalculated and therefore have limited length. More information on using blue noise to improve sampling can be found in Chapter 24.

14.3.5 ACCUMULATION AND ANTIALIASING

With the ability to trace rays and a dependable random number generator in hand, we now have the tools necessary to sample a scene, accumulate the results, and progressively refine an image. Shown in Listing 14-6, progressive refinement is implemented by adding the image generated by each frame to an accumulation buffer and then normalizing the accumulated colors before displaying the image shown on screen. Figure 14-6 compares a single frame's result with the normalized accumulated result from many frames.

Listing 14-6. *Implementing an accumulation buffer (ray generation shader).*

```
uint2 LaunchIndex = DispatchRaysIndex().xy;
// Trace a path for the current pixel.
// ...
// Get the accumulated color.
float3 previousColor = accumulationBuffer[LaunchIndex].rgb;

// Add the current image's results.
float3 accumulatedColor = previousColor + radiance;
accumulationBuffer[LaunchIndex] = float4(accumulatedColor, 1.f);

// Normalize the accumulated results to get the final result.
return linearToSrgb(accumulatedColor / gData.accumulatedFrames);
```



Figure 14-6. *A single path traced frame (left) and the normalized accumulated result from one million frames (right).*

The number of accumulated frames is used to ensure that each frame is weighted equally when converting the accumulation buffer's contents to the final image. We use a high-precision 32-bit floating-point format for our accumulation buffer to allow for a large number of samples to be included. Note that typical render target formats in the rasterization pipeline can depend on hardware auto-conversion from linear to sRGB color spaces when writing results. This is not currently supported for the unordered access view types commonly used for output buffers in ray tracing, so it may be necessary to manually convert the final result to sRGB. This is implemented in our code sample by the `linearToSrgb` function.

In addition to amortizing the high cost of computing paths, accumulating images generated by randomly sampling the scene also naturally antialiases the final result. Shown in Listing 14-7, antialiasing is now trivial to implement by randomly jittering ray directions (of primary rays as well as rays traced for material evaluation). For primary rays, this robust sampling not only removes jagged edges caused by geometric aliasing, but also mitigates the moiré patterns that appear on undersampled textured surfaces.

Listing 14-7. *Jittering primary ray directions for antialiasing.*

```
    float2 pixel = float2(DispatchRaysIndex().xy);
    float2 resolution = float2(DispatchRaysDimensions().xy);

    // Add a random offset to the pixel's screen coordinates.
    float2 offset = float2(rand(rngState), rand(rngState));
    pixel += lerp(-0.5.xx, 0.5.xx, offset);

    pixel = (((pixel + 0.5f) / resolution) * 2.f - 1.f);
    generatePinholeCameraRay(pixel);
```

This convenient property of accumulation makes it possible to forgo texture level of detail strategies (such as mipmapping and ray differentials) and sample *only* the highest-resolution version of any texture. Since current graphics hardware and APIs do not support automatic texture level of detail mechanisms in ray tracing, the benefits of accumulation make it possible for our reference path tracer to avoid the added code and complexity related to implementing it (at the cost of performance). Should improved performance and support for texture level of detail become necessary (e.g., to implement ray traced reflections in the real-time renderer), suitable methods are available in *Ray Tracing Gems* [2], *The Journal of Computer Graphics Techniques* [1], and Chapters 7 and 10 of this book.

```
ray = generatePrimaryRay();
throughput = 1.0;
radiance = 0.0;
for bounce ∈ {1 . . . MAX_BOUNCES} do
    Trace(ray);
    if hit surface then
        brdf, brdfPdf, ray = SampleBrdf();
        throughput *= brdf / brdfPdf;
    else
        radiance += throughput * skyColor;
        break;
return radiance;
```

Figure 14-7. *The basic path tracing loop.*

14.3.6 TRACING PATHS

Now we are ready to trace rays beyond primary rays and create full paths from the camera to surfaces and lights in the environment. To accomplish this, we extend our existing ray generation shader and implement a basic path tracing loop, illustrated in Figure 14-7. The process begins by initializing a ray to be cast from the camera into the environment (i.e., a primary ray). Next, we enter a loop and ray tracing begins. If a ray trace misses all surfaces in the scene, the sky's contribution is added to the result color and the loop is terminated. If a ray intersects geometry in the scene, the intersected surface's material properties are loaded and the associated bidirectional reflectance distribution function (BRDF) is evaluated to determine the direction of the next ray to trace along the path. The BRDF accounts for the composition of the material, so for rough surfaces, the reflected ray direction is randomized and then attenuated based on object color. Details on BRDF evaluation are described in the next section.

For a simple first test of the path tracing loop implementation, set the sky's contribution to be fully white (`float3(1, 1, 1)`). This creates lighting conditions commonly referred to as the *white furnace* because all surfaces in the scene are illuminated with white light equally from all directions. Shown in Figure 14-8, this lighting test is especially helpful when evaluating the energy conservation characteristics of BRDF implementations. After the



Figure 14-8. *The interior scene rendered using the white furnace test with white diffuse materials applied to all surfaces.*

white furnace test, try loading values from an environment map in place of a fixed sky color.

Note the two variables, radiance and throughput, declared and initialized at the beginning of the path tracing loop in Figure 14-7. *Radiance* is the final intensity of the light energy presented on screen for a given path. Radiance is initially zero and increases as light is encountered along the path being traced. *Throughput* represents the amount of energy that may be transferred along a path's ray segment after interacting with a surface. Throughput is initialized to one (the maximum) and decreases as surfaces are encountered along the path. Every time the path intersects a surface, the throughput is attenuated based on the properties of the intersected surface (dictated by the material BRDF at the point on the surface). When a path arrives at a light source (i.e., an emissive surface), the throughput is *multiplied* by the intensity of the light source and *added* to the radiance. Note how simple it is to support emissive geometry using this approach!

SURFACE MATERIALS

To render a scene with a variety of realistic materials, we implement BRDFs that describe how light interacts with surfaces. Most surfaces in the physical world reflect some subset of the incoming light (or light wavelengths) in a set

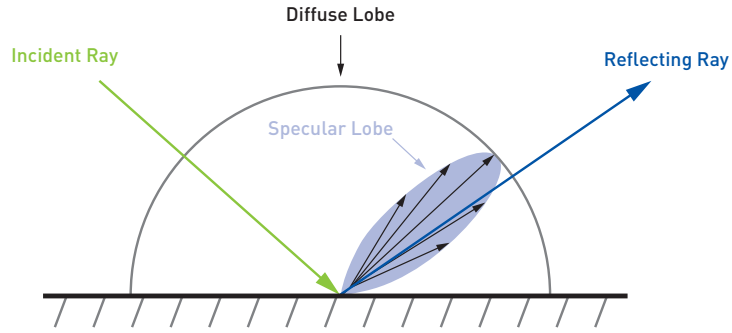


Figure 14-9. *Importance sampling of a BRDF involves selecting a direction of the reflecting ray, based on the incident ray direction and surface properties. The image shows several possible ray directions for the specular lobe (black arrows) and one selected direction highlighted in blue.*

of directions defined by the surface's micro-scale geometric composition. These parameters of variation create an incredible amount of distinct colors and appearances. To simulate this behavior and achieve a desired surface appearance, we (a) attenuate ray throughput according to a material BRDF and (b) reflect incident light within the solid angle determined by the material BRDF (often called a *lobe*). For example, metals and mirrors reflect most incoming light in the direction of perfect reflection, whereas rough matte surfaces reflect light in all directions over the hemisphere above the surface.

To properly attenuate the ray throughput, we importance-sample the material's BRDF lobes by evaluating the *brdf* and *brdfPdf* values shown in Figure 14-7. Importance sampling ensures that each new ray along the path contributes in a meaningful way to the evaluation of the surface material [9]. *brdfPdf* is a probability density function (PDF) of selecting a particular ray direction over all other possible directions, and it is specific to the importance sampling method being used. In our sample code, this process is encapsulated in the `sampleBrdf` function. This function generates a direction for the next ray to be traced along a path and computes the BRDF value corresponding to the given combination of incident and reflected ray directions. See Figure 14-9.

To ease into BRDF implementation, we recommend starting with diffuse materials. In this case, incoming light is reflected equally in all directions, which is simpler to implement. We implement this behavior by generating ray directions within a cosine weighted hemisphere (see `sampleHemisphere` in

the sample code) and using a PDF of *brdfPdf* = $\cos\omega/\pi$. This distributes rays across the entire hemisphere above the surface proportional to the cosine term found in the rendering equation. The Lambertian term *brdfWeight* = *diffuseColor*/π can be pre-divided by the PDF and multiplied by the cosine term, resulting in *brdfWeight*/*brdfPdf* being equal to the diffuse color of the surface.

To support a wide range of physically based materials, we employ a combination of specular and diffuse BRDFs commonly used in games as detailed by Boksansky [4]. Note that these implementations return BRDF values already divided by the PDF, while also accounting for the rendering equation's cosine term, so these calculations are not explicitly shown in our code sample.

SELECTING BRDF LOBES

Many material models are composed of two or more BRDF lobes, and it is common to evaluate separate lobes for the specular and diffuse components of a material. In fact, when rendering semitransparent or refractive objects, an additional *bidirectional transmittance distribution function* (BTDF) is also evaluated. This is discussed in more detail in Chapter 11.

Since the reference path tracer accumulates images, we are able to progressively sample all lobes of a material's BRDF while only tracing a single ray from each surface every frame. With this comes a new choice: *which* BRDF lobe should be sampled on each intersection? A simple solution is to randomly choose one of the lobes, each with equal probability; however, this may sample a component of the overall BRDF that ultimately doesn't contribute much to the final result. For example, highly reflective metals do not have a diffuse component, so the diffuse lobe does not need to be sampled at all.

To address this, we (a) evaluate a subset of each lobe's terms ahead of selection to use to (b) estimate the lobe's contribution and then (c) use the estimate to set the probability of selecting the lobe. Lobes with higher estimated contributions receive higher probabilities and are selected more often. This approach requires a careful balance between the number of terms evaluated before selection and the efficiency of the entire selection process. Note that some terms can only be evaluated once the reflected ray direction is known, which is expensive to generate.

```
pSpecular = getBrdfProbability();
if rand(rngState) < pSpecular) then
    brdfType = SPECULAR;
    throughput /= pSpecular;
else
    brdfType = DIFFUSE;
    throughput /= (1 - pSpecular);
```

Figure 14-10. *Importance sampling and BRDF lobe selection. We select lobes using a random number and probability $p_{specular}$. Throughput is divided by either $p_{specular}$ or $1 - p_{specular}$, depending on the selected lobe.*

In our code sample, we implement BRDF lobe selection using the importance sampling algorithm shown in Figure 14-10. The probability of choosing a specular or diffuse lobe as an estimate of their respective contributions e_{specular} and e_{diffuse} is written as

$$p_{specular} = \frac{e_{\text{specular}}}{e_{\text{specular}} + e_{\text{diffuse}}}, \tag{14.1}$$

where e_{specular} is equal to the Fresnel term F. We evaluate F using the shading normal instead of the microfacet normal (i.e., the view and reflection direction half-vector), since the true microfacet normal is known only after the reflected ray direction is computed by sampling the specular lobe. This works reasonably well for most materials, but is less efficient for rough surfaces viewed at grazing angles. Note that estimate e_{diffuse} is only based on the diffuse reflectance of the surface. Depending on the way a material model combines diffuse and specular lobes together, e_{diffuse} may also need to be weighted by $1 - F$.

This approach is simple and fast, but can generate lobe contribution estimates that are very small in some cases (e.g., rough materials with a slight specular highlight or an object that is barely semitransparent). Low-probability estimates cause certain lobes to be undersampled and can manifest as a "salt and pepper" pattern in the image. Dividing by small probability values introduces numerical precision issues and may also introduce firefly artifacts. To solve these problems, we clamp p_{specular} to the range $[0.1, 0.9]$ whenever it is not equal to zero or one. This ensures a reasonable minimal sampling frequency for each contributing BRDF lobe when the estimates are very small.

TERMINATING THE PATH TRACING LOOP

Since we are tracing paths by iteratively casting rays, we need to handle termination of the loop. Light paths naturally end when the light energy carried along the path is fully absorbed or once the path exits the scene to the sky, but reaching this state can require an extremely large number of bounces. In closed interior scenes, paths may never exit the scene. To prevent tracing paths of unbounded length, we define a maximum number of bounces and terminate the ray tracing loop once the maximum is reached, as shown in Figure 14-7. Bias is introduced when terminating paths too early, but paths that never encounter light are unnecessary and limit performance.

To balance this trade-off, we use a *Russian roulette* approach to decide if insignificant paths should be terminated early. An insignificant path is one that encounters a surface that substantially decreases ray throughput, and as a result will not significantly contribute to the final image. Shown in Listing 14-8, Russian roulette is run before tracing each non-primary ray, and it randomly selects whether to terminate the ray tracing loop or not. The probability of termination is based on the luminance of the path's throughput and is clamped to be at most 0.95 so every possible surface interaction, even those with perfect mirrors, may be terminated. To avoid introducing bias from path termination, the throughput of the non-terminated path is increased using the termination probability. We allow Russian roulette termination to start after a few initial bounces (three in our code sample), to prevent light paths from being terminated too early.

Listing 14-8. *Using a Russian roulette approach to decide if paths should terminate.*

```
    if (bounce > MIN_BOUNCES) {
        float rr_p = min(0.95f, luminance(throughput));
        if (rr_p < rand(rngState)) break;
        else throughput /= rr_p;
    }
```

Integration of Russian roulette into the path tracing loop is shown in Figure 14-11. This improvement allows us to increase the maximum number of bounces a path may have and ensures computation time is spent primarily where it is needed.

SURFACE NORMALS

Since ray segments along a path may intersect a surface on either side—not only the side facing the camera—it is best to disable backface culling when

```
Initialization...
for bounce ∈ {1 . . . MAX_BOUNCES} do
    Trace ray and evaluate direct lighting...
    russianRoulette = luminance(throughput);
    if russianRoulette < rand() then
        break;
    else
        throughput /= russianRoulette;
return radiance;
```

Figure 14-11. *The improved path tracing loop with Russian roulette path termination.*

tracing rays (which also improves performance). As a result, special care is required to ensure that correct surface normals are available on both sides of the surface. Shown in Listing 14-9, we account for this programmatically by inverting normals with the same direction as the incident ray.

Listing 14-9. *Inversion of backfacing normals.*

```
1    float3 V = -ray.Direction;
2    if (dot(geometryNormal, V) < 0.0f) geometryNormal *= -1;
3    if (dot(geometryNormal, shadingNormal) < 0.0f) shadingNormal *= -1;
```

A related issue is self-intersections that occur when a ray intersection is reported for the same surface from which a ray originates. This happens when the ray origin is erroneously placed on the opposite side of a surface due to insufficient numerical precision. A simple solution to this problem is to move the origin away from the surface along the normal by a small fixed amount ϵ ($\epsilon \approx 0.001$), but this approach is not general and almost always introduces light leaking and disconnected shadows. To prevent these problems, we use the improved approach described in Chapter 6 of *Ray Tracing Gems* [16], which is based on calculating the minimal offset necessary to prevent self-intersections (see Figure 14-12). Note that this approach works best when the ray/surface intersection position is calculated from the triangle's vertex positions and interpolated with barycentrics, as this is more precise than using the ray origin, direction, and hit distance. This method also allows us to set the `TMin` of reflecting rays to zero and prevent self-intersections by simply updating the ray origin.

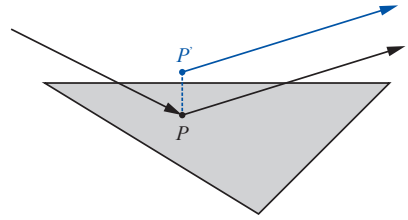


Figure 14-12. *The `offsetRay` method in our code sample calculates a new ray origin P′ originating at the ray/surface intersection point P to minimize self-intersections and light leaks caused by insufficient numerical precision.*

14.3.7 VIRTUAL LIGHTS AND SHADOW RAYS

Illumination from the sky produces beautiful results in a variety of exterior and open interior scenes, but emissive objects (i.e., lights!) are another important component when illuminating interior scenes. In this section, we cover virtual lights and shadow rays. This special treatment of light sources is sometimes referred to as *next event estimation*.

To support virtual lights, we modify the path tracing loop as shown in Figure 14-13. At every ray/surface intersection we randomly select one light (see `sampleLight` in the code sample) and cast a *shadow ray* that determines the visibility between the light and the surface. If the light is visible from the surface, we evaluate the surface's BRDF for the light, divide by the sampling PDF of the light, and add the result multiplied by the throughput to the path radiance (similar to how we handle emissive geometry). This process is illustrated in Figure 14-14.

Since point lights are just a position in space without any area, the contribution of the point light increases to *infinity* as the distance between the light and a surface approaches zero. This creates a singularity that causes invalid values (NaNs) and firefly artifacts that are important to mitigate in a path tracer. We use a method by Yuksel to avoid this obstacle when evaluating point lights [20]. Listing 14-10 shows how point light evaluation is implemented in our sample code. Our code sample supports point and directional lights, but it is straightforward to implement more light types as necessary. For debugging use, we automatically place one directional light source (the sun) and one point light source attached to the camera (a headlight) in the scene. To improve performance when evaluating lights, we

```
Initialization...
for bounce ∈ {1 . . . MAX_BOUNCES} do
    Trace(ray);
    if hit then
        lightIntensity, lightPdf = SampleLight();
        radiance += ray.throughput * EvalBrdf() * lightIntensity *
          CastShadowRay() / lightPdf;
        if bounce == MAX_BOUNCES then
            break;
        brdf, brdfPdf, ray = SampleBrdf();
        throughput *= brdf / brdfPdf;
    else
        radiance += throughput * skyColor;
        break;
    Russian roulette ray termination...
return radiance;
```

Figure 14-13. *The path tracing loop modified to support virtual lights.*

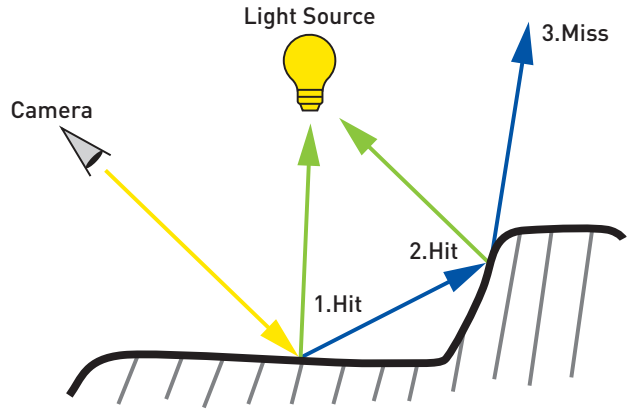


Figure 14-14. *An illustration of the path tracing loop with virtual lights. A primary ray (yellow) is generated and cast. At every surface intersection, a shadow ray (green) is cast toward selected light sources in addition to the BRDF lobe ray (blue).*

create a dedicated DXR hit group for shadow ray tracing—as these rays only need to determine visibility—and skip the closest-hit shader in the ray tracing pipeline.

Listing 14-10. *Evaluating virtual lights in a path tracer.*

```
Light light;
float lightWeight;
sampleLight(rngState, hitPosition, geometryNormal, light, lightWeight);

// Prepare data needed to evaluate the light.
float lightDistance = distance(light.position, hitPosition);
float3 L = normalize(light.position - hitPosition);

// Cast shadow ray toward the light to evaluate its visibility.
if (castShadowRay(hitPosition, geometryNormal, L, lightDistance))
{
    // Evaluate BRDF and accumulate contribution from sampled light.
    radiance += throughput * evalCombinedBRDF(shadingNormal, L, V, material)
          * (getLightIntensityAtPoint(light, lightDistance) * lightWeight);
}
```

SELECTING LIGHTS

In the majority of situations, it is not practical to evaluate and cast shadow rays from all surfaces to all virtual lights in a scene. In the previous section we depended on the `sampleLight` function to select one virtual light from the set of all lights in the scene at each ray/surface intersection. But what is the "correct" light to select and how should this work?

Shown in Listing 14-11, a simple solution is to randomly select one light from the list of lights using a uniform distribution, with a probability of selection equal to 1/*N*, where *N* is number of lights (the `lightWeight` variable in Listing 14-10 is equal to reciprocal of this PDF, analogous to the way we handle BRDF sampling). This approach is straightforward but produces noisy results because it does not consider the brightness of or distance to the selected light. On the opposite end of the spectrum, an expensive solution may evaluate the surface's BRDF for all lights to establish the probability of selecting each light, and then use importance sampling to select lights based on their actual contribution.

Listing 14-11. *Random selection of a light from all lights with a uniform distribution.*

```
uint randomLightIndex =
        min(gData.lightCount - 1, uint(rand(rngState) * gData.lightCount));
light = gData.lights[randomLightIndex];
lightSampleWeight = float(gData.lightCount);
```

A reasonable compromise between these options is a method based on resampling introduced by Talbot [15]. Instead of evaluating the full surface BRDF for all lights, we randomly select a few candidates (e.g., around eight) with a simple uniform distribution method. Next, we evaluate either the full or partial BRDF for these candidate lights and stochastically choose one light based on its contribution. With this approach, we are able to quickly discard lights on the backfacing side of the surface and select ones that are visible, brighter, and closer more often. This method is included in our code sample in the `sampleLightRIS` function. More details about light selection and resampling can be found in Chapters 22 and 23.

TRANSPARENCY

The rendering of transparent and translucent surfaces is a classic challenge in graphics, and real-time ray tracing provides us with powerful new tools to improve existing real-time techniques for transparency. In our code sample and shown in Listing 14-12, we include a simple *alpha-test* transparency mode where a surface's alpha value is compared to a specified threshold and intersections are ignored depending on the result. Any-hit shaders are invoked on every intersected surface while searching for the closest hit, which gives us the opportunity to ignore the hit if the surface's alpha is below the transparency threshold. To perform this test, the alpha value often comes from a texture and must be loaded on *all* any-hit shader invocations. This operation is expensive, so it is best to ensure that any-hit shaders are only executed for geometry with transparent materials. To do this in DXR, mark all BLAS geometry representing fully opaque surfaces with the `D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQUE` flag.

Listing 14-12. *Implementation of the alpha test transparency using the any-hit shader. Note the relatively large number of operations performed on every tested surface while searching for the closest hit.*

```
MaterialData mData = materials[materialID];
float opacity = mData.opacity;

if (mData.baseColorTexIdx != -1) {
    float3 barycentrics = float3((1.0f - attrib.uv.x - attrib.uv.y), attrib.
        uv.x, attrib.uv.y);
    VertexAttributes vertex = GetVertexAttributes(geometryID, PrimitiveIndex
        (), barycentrics);
    opacity *= textures[mData.baseColorTexIdx].SampleLevel(linearSampler,
        vertex.uv, 0.0f).a;
}

if (opacity < mData.alphaCutoff) IgnoreHit();
```

The alpha-tested transparency discussed here barely scratches the surface of the complexity that transparency introduces to the rendering process. This topic deserves an entire chapter (or book!) of its own, and you can learn more about rendering transparent surfaces in Chapter 11.

14.3.8 DEFOCUS BLUR

Defocus blur, sometimes called a *depth of field* or *bokeh*, is an interesting visual effect that is difficult to implement with rasterization-based renderers, yet easy to achieve with a path tracer. To implement it, we generate primary rays in a way that emulates how light is transported through a real camera lens and then accumulate many frames over time to smooth out the results. Our code sample contains a simple implementation of this effect using a thin lens model (see the `generateThinLensCameraRay` function). More details about defocus blur can be found in Chapter 1 and *Physically Based Rendering* [11].

14.4 CONCLUSION

The path tracer described in this chapter is relatively simple, but suitable for rendering high-quality references for games and as an experimentation platform for the development of real-time effects. It has been used during the development of an unreleased AAA game and has proved to be a helpful tool. The code sample accompanying this chapter is freely available, contains a functional path tracer, and can be extended and integrated into game engines without any restrictions.

There are many ways this path tracer can be improved, including support for refraction, volumetrics, and subsurface scattering as well as denoising techniques and performance optimizations to produce noise-free results while running at real-time frame rates. The accompanying chapters in this book, its previous volume [6], and *Physically Based Rendering* [11] are excellent sources of inspiration for improvements as well. We recommend the blog posts "Effectively Integrating RTX Ray Tracing into a Real-Time Rendering Engine" [14] and "Optimizing VK/VKR and DX12/DXR Applications Using Nsight Graphics: GPU Trace Advanced Mode Metrics" [3] for guidance on the integration of real-time ray tracing into existing engines and on optimizing performance.

REFERENCES

[1] Akenine-Möller, T., Crassin, C., Boksansky, J., Belcour, L., Panteleev, A., and Wright, O. Improved Shader and Texture Level of Detail Using Ray Cones. *Journal of Computer Graphics Techniques*, 10(1):1–24, 2021. http://jcgt.org/published/0010/01/01/.

[2] Akenine-Möller, T., Nilsson, J., Andersson, M., Barré-Brisebois, C., Toth, R., and Karras, T. Texture Level of Detail Strategies for Real-Time Ray Tracing. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 20, pages 321–345. Apress, 2019.

[3] Bavoil, L. Optimizing VK/VKR and DX12/DXR Applications Using Nsight Graphics: GPU Trace Advanced Mode Metrics. *NVIDIA Developer Blog*, https://developer.nvidia.com/blog/optimizing-vk-vkr-and-dx12-dxr-applications-using-nsight-graphics-gpu-trace-advanced-mode-metrics, March 30, 2020. Accessed March 17, 2021.

[4] Boksansky, J. Crash Course in BRDF Implementation. https://boksajak.github.io/blog/BRDF, February 28, 2021. Accessed March 17, 2021.

[5] Evermotion. 3D Archinteriors Vol. 48 for Blender. https://www.turbosquid.com/3d-models/3d-archinteriors-vol-48-blender-1308896, 2021. Accessed March 17, 2021.

[6] E. Haines and T. Akenine-Möller, editors. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress, 2019.

[7] Jarzynski, M. and Olano, M. Hash Functions for GPU Rendering. *Journal of Computer Graphics Techniques*, 9(3):20–38, 2020. http://jcgt.org/published/0009/03/02/.

[8] Jenkins, B. Hash Functions. *Dr. Dobb's*, https://www.drdobbs.com/database/algorithm-alley/184410284, September 1, 1997. Accessed March 17, 2021.

[9] Kahn, H. and Marshall, A. Methods of Reducing Sample Size in Monte Carlo Computations. *Journal of the Operations Research Society of America*, 1(5):263–278, 1953. DOI: 10.1287/opre.1.5.263.

[10] Marrs, A. Introduction to DirectX Raytracing. https://github.com/acmarrs/IntroToDXR, 2018. Accessed March 17, 2021.

[11] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, third edition, 2016.

[12] Reed, N. Quick and Easy GPU Random Numbers in D3D11. http://www.reedbeta.com/blog/quick-and-easy-gpu-random-numbers-in-d3d11/, January 12, 2013. Accessed March 17, 2021.

[13] Shirley, P. Ray Tracing in One Weekend. https://raytracing.github.io/books/RayTracingInOneWeekend.html, 2018. Accessed March 17, 2021.

[14] Sjoholm, J. Effectively Integrating RTX Ray Tracing into a Real-Time Rendering Engine. *NVIDIA Developer Blog*, https://developer.nvidia.com/blog/effectively-integrating-rtx-ray-tracing-real-time-rendering-engine, October 29, 2018.

[15] Talbot, J., Cline, D., and Egbert, P. Importance Resampling for Global Illumination. In *Eurographics Symposium on Rendering*, pages 139–146, 2005. DOI: 10.2312/EGWR/EGSR05/139-146.

[16] Wächter, C. and Binder, N. A Fast and Robust Method for Avoiding Self-Intersection. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 6, pages 77–85. Apress, 2019.

[17] Wang, T. Integer Hash Function. http://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm, 1997. Accessed March 17, 2021.

[18] Wyman, C., Hargreaves, S., Shirley, P., and Barré-Brisebois, C. Introduction to DirectX Raytracing. *ACM SIGGRAPH 2018 Courses*, http://intro-to-dxr.cwyman.org, 2018.

[19] Wyman, C. and Marrs, A. Introduction to DirectX Raytracing. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 3, pages 21–47. Apress, 2019.

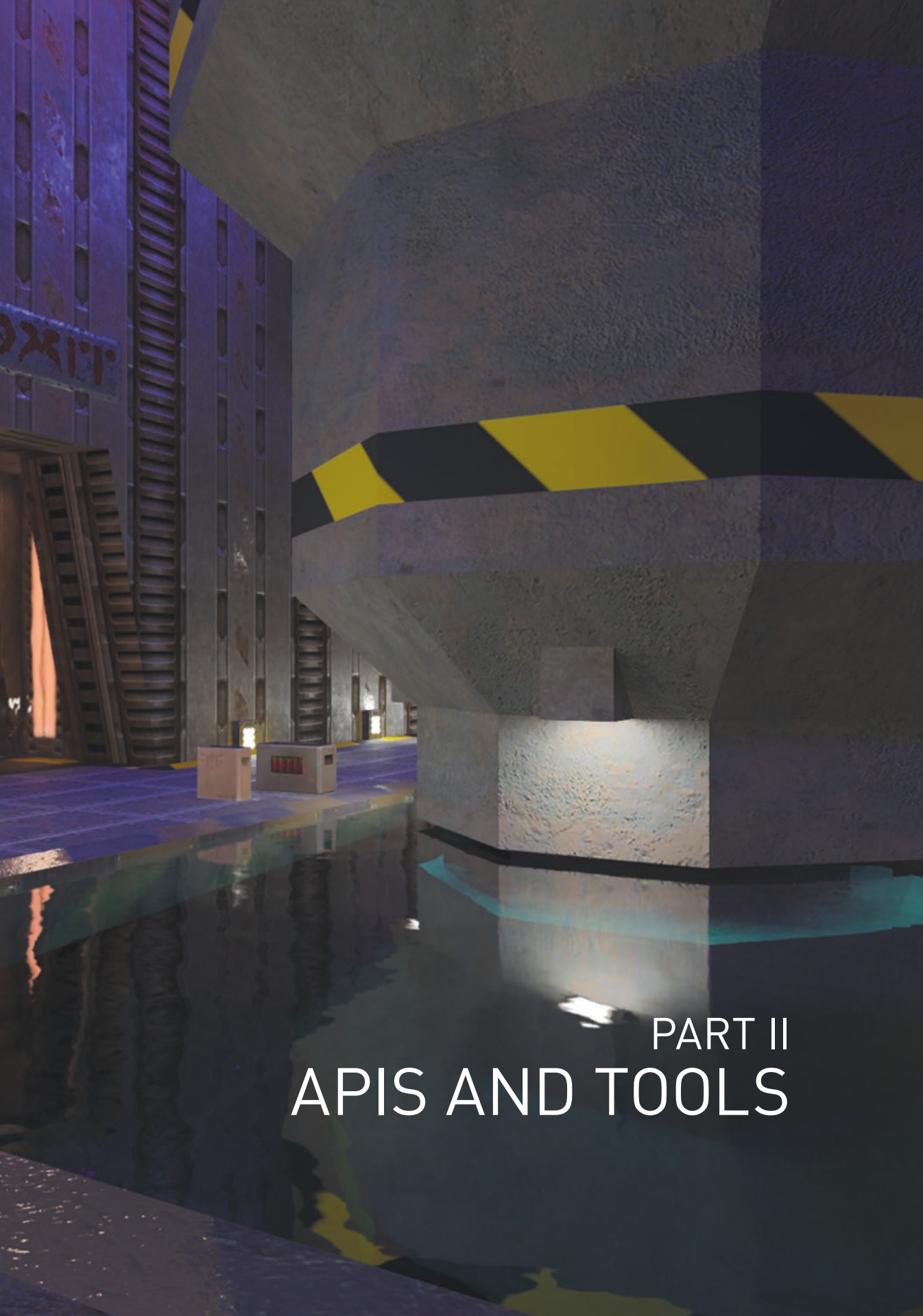
[20] Yuksel, C. Point Light Attenuation Without Singularity. In *ACM SIGGRAPH 2020 Talks*, 18:1–18:2, 2020. DOI: 10.1145/3388767.3407364.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





PART II
APIS AND TOOLS

PART II

APIS AND TOOLS

Since NVIDIA RTX technology introduced ray tracing hardware acceleration to consumer GPUs, the number of games and applications supporting ray tracing has skyrocketed, along with the number of developers that need to interface with one of the standardized ray tracing APIs like DirectX, Vulkan, or OptiX. This part is devoted to practical introductions to some of the APIs and tools that are available to help today's developers learn and apply ray tracing in their work.

Chapter 15, *The Shader Binding Table Demystified*, explains succinctly how one of the trickiest parts of today's ray tracing APIs works. The shader binding table is notorious for causing confusion among developers new to ray tracing on the GPU. This chapter covers in detail what the shader binding table really is, what it can be used to do, and how the APIs compute indices into the table at runtime. Finally, this chapter walks through some concrete examples of common use cases and ways to set up the shader binding table to achieve them.

Chapter 16, *Introduction to Vulkan Ray Tracing*, gives the reader a complete introduction to ray tracing using the Vulkan API's ray tracing extensions. This chapter is similar in spirit to the "Introduction to DirectX Raytracing" chapter from the first *Ray Tracing Gems* book. Since the first book was published, the Khronos Group has introduced ray tracing support into the Vulkan API via a set of extensions that provide an API familiar to users of DirectX Raytracing and OptiX. These Vulkan extensions provide a standardized and cross-platform way to access hardware-accelerated ray tracing, as well as integrate seamlessly with Vulkan rasterization-based applications. Source code for a complete working example application is provided.

Chapter 17, *Using Bindless Resources with DirectX Raytracing*, shows how to avoid the explicit resource binding patterns that have been common with DirectX rasterization. With bindless techniques, shaders have access to all loaded textures and other resources, while also using simpler code. Because rays can randomly intersect anything in the scene, global random access to resources when ray tracing is essential. In some cases, the new bindless

patterns are the only practical approach to ray tracing with DirectX. A complete source code example of a bindless path tracer is included.

Chapter 18, *WebRays: Ray Tracing on the Web*, introduces a new API built for doing ray tracing in a web browser, expanding hardware-accelerated ray tracing access to web applications. WebRays is currently powering the website Rayground, which allows people to both develop and run ray tracing applications in their web browser. WebRays supports shaders written in GLSL via WebGL and provides a JavaScript host-side API and a GLSL device-side API to enable hardware-accelerated ray/triangle intersections. This ray casting API has been designed to allow both wavefront and megakernel style architectures, and it tries to make room for any style of ray tracing pipeline the user wants.

Chapter 19, *Visualizing and Communicating Errors in Rendered Images*, offers developers a useful tool and a framework for evaluating how well their rendering algorithms are performing in terms of image quality. The ꟻLIP tool implements a perceptual error metric that is modeled after the way many experts evaluate their images: by flipping back and forth between an old image and a new one to watch for which parts of the image change. This chapter covers how to measure render error in high dynamic range images, as well as low dynamic range images, and discusses how to read, think about, and analyze these kinds of measurements. Finally, the chapter applies this methodology to the evaluation of a new example rendering algorithm to show how to evaluate and choose rendering parameters. The tool is open source and comes with implementations for both the CPU and the GPU.

David Hart

CHAPTER 15

THE SHADER BINDING TABLE DEMYSTIFIED

Will Usher
Intel Corporation

ABSTRACT

When a ray hits a geometry in the scene, the ray tracing hardware needs some way to determine which shader to call to perform intersection tests or shading calculations. The Shader Binding Table (SBT) provides this information. The SBT is a lookup table that associates each geometry in the scene with a set of shader function handles and parameters for these functions. Each set of function handles and parameters is referred to as a *shader record*. When a geometry is hit by a ray, a set of parameters specified on both the host and device sides of the application combine to determine which shader record is called. The three GPU ray tracing APIs (DirectX Raytracing, Vulkan KHR Ray Tracing, and OptiX) use the SBT to run user code when tracing rays in the scene. These APIs provide a great deal of flexibility as to how the SBT can be set up and indexed during rendering, providing a number of options to applications. However, this generality makes properly setting up the SBT an especially thorny part of these APIs for new users, as incorrect SBT access often leads to crashes or difficult bugs. Translating knowledge from one API to another can also be difficult due to subtle naming differences between them. This chapter looks at the similarities and differences of each API's SBT to develop a holistic understanding of the programming model. A few common configurations of the scene, SBT, and renderer are also discussed to provide guidance through hands-on examples.

15.1 THE SHADER BINDING TABLE

The Shader Binding Table (SBT) contains the entire set of shaders that may be called when ray tracing the scene, along with embedded parameters to be passed to these shaders when they are called. Each set of shader function handles and embedded parameters is referred to as a *shader record*. There are three different types of shader records that can be stored in the SBT: ray

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_15

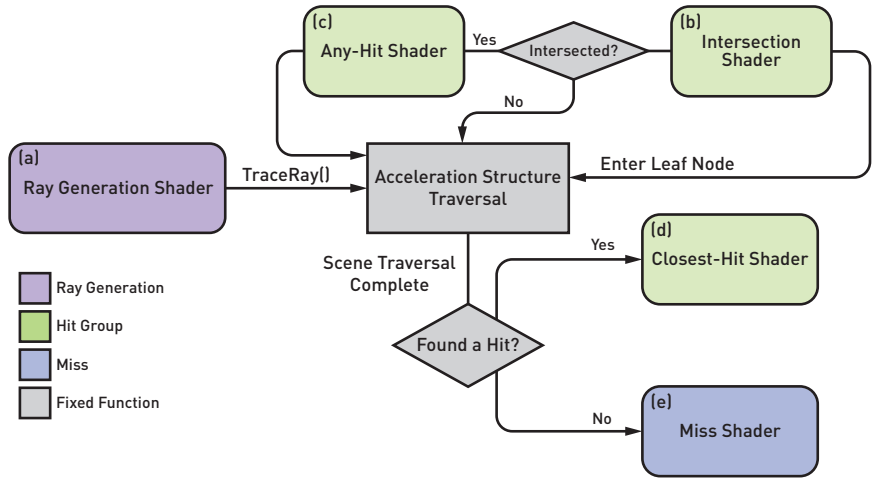


Figure 15-1. *An illustration of the ray tracing pipeline. (a) The ray generation shader computes and traces rays by calling* `TraceRay`*. During traversal, rays are tested against primitives in the leaf nodes of the acceleration structure. (b) The intersection shader is called to compute intersections when using custom primitives. (c) The any-hit shader is called if an intersection is found, if the any-hit shader is enabled. After scene traversal is complete, (d) the closest-hit shader of the intersected geometry is called if a hit was found and the closest-hit shader is enabled. (e) The miss shader is called if no intersection was found. The closest-hit and miss shaders can then either return control to the caller of* `TraceRay` *or make recursive trace calls.*

generation records, hit group records, and miss records. Each *ray generation record* provides the handle of a ray generation shader, which acts as the entry point to the ray tracer and is typically responsible for generating and tracing rays. *Hit group records* provide shader functions to be called when a geometry in the scene is hit by a ray. A hit group record can contain three different shader function handles, the *intersection* shader, *any-hit* shader, and *closest-hit* shader, which are called at different stages of the ray tracing pipeline during ray traversal. *Miss records* provide a shader to be called when the ray does not hit any geometry. A ray tracer may have several *ray types*: for example, primary and shadow rays. Different hit group and miss records are typically provided for each ray type. For example, though the intersection shader must find the closest intersection of a primary ray with the geometry, for shadow rays it is sufficient to determine that any intersection occurred, possibly making such tests faster. Figure 15-1 illustrates where the different shaders are called in the ray tracing pipeline.

15.1.1 RAY GENERATION RECORDS

A ray generation record consists of a single shader function handle for the ray generation shader, along with any embedded parameters for the shader. The ray generation shader acts as the entry point to the renderer, and typically generates and traces rays into the scene to render it (Figure 15-1a). Each API also provides a way for shaders to access a set of global parameters. Global or frequently updated parameters can be passed through these global parameters, whereas other static data can be passed directly in the ray generation record's parameters. Although multiple ray generation records can be written into the SBT, only one can be called for any individual launch.

15.1.2 HIT GROUP RECORDS

A hit group record can specify three shader function handles along with embedded shader parameters for them. The embedded parameters are shared by the three shaders. These shaders are the following:

- *Intersection:* The intersection shader is required when using custom geometry and is responsible for computing ray intersections with the custom primitives. It is called when a ray intersects the custom geometry's bounding box to test it for intersection with the geometry (Figure 15-1b). The intersection shader is not required when using built-in primitive types.

- *Any-hit:* The any-hit shader is called if an intersection is found with the geometry and can be used to filter out undesired intersections (Figure 15-1c). For example, when using alpha cutout textures, the any-hit shader is used to discard intersections with the cutout regions of the geometry.

- *Closest-hit:* The closest-hit shader is called when ray traversal has completed and an intersection was found along the ray (Figure 15-1d). The closest-hit shader is typically used to perform shading of the geometry or return its surface properties to the caller, and it can trace additional rays.

All three shaders in the hit group are optional and can be specified or left empty depending on the geometry with which the hit group is associated and the ray flags that will be specified when tracing the ray types for which the hit group will be called. The hit group that is called for a specific geometry

depends on multiple parameters specified when creating the scene geometry and tracing the ray, and are discussed in Section 15.2.1.

15.1.3 MISS RECORDS

A miss record consists of a single shader function handle, the *miss shader*, and any embedded parameters for the shader. The miss shader is called when the ray did not hit an object in the scene (Figure 15-1e). One miss record is typically provided per ray type. For example, for primary rays the miss shader can return a background color, while for shadow rays it can mark the ray as unoccluded. However, the miss record ray type is specified separately from the hit group ray type when tracing a ray to provide more flexibility; this can be used to implement optimizations for shadow rays (see Section 15.4.3).

15.2 SHADER RECORD INDEX CALCULATION

The main point of difficulty in setting up the SBT and scene geometry is seeing how the two are coupled together; i.e., if a geometry is hit by a ray, which shader record is called? The shader record to call is determined by parameters set on the instance, the trace ray call, and the order of geometries in the bottom-level acceleration structure containing it. These parameters are set on both the host and device during different parts of the scene setup and pipeline execution, making it difficult to see how they interact.

15.2.1 HIT GROUP RECORDS

The scene being rendered can contain multiple instances, where each one references a *bottom-level acceleration structure* (BLAS). Each BLAS contains an array of one or more geometries, with each geometry made up of an array of built-in triangles, other built-in primitives, or axis-aligned bounding boxes for custom primitives. The index of a geometry in the BLAS's array of geometries is its geometry ID, $\mathbb{G}_{\text{ID}}$, which can be used to offset into the hit group record array in the SBT. Each instance can specify a starting offset in the hit group record array, $\mathbb{I}_{\text{offset}}$, to specify where the sub-array of hit group records for its BLAS's geometries starts.

Additional parameters can be specified when calling `TraceRay` that affect hit group indexing: an additional SBT offset for the ray, R_{offset}, typically referred to as the ray type, and an SBT stride to apply between each geometry's hit group records, R_{stride}, typically referred to as the number of ray types. Note that the DirectX Raytracing chapter in *Ray Tracing Gems* [12] uses a slightly

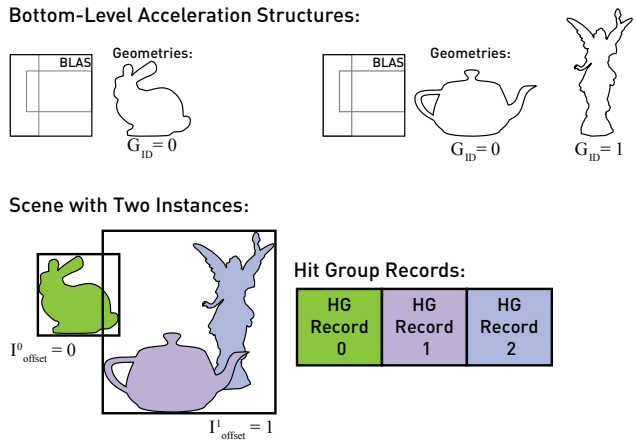


Figure 15-2. *The hit group records called for each geometry in a simple ray tracer with a single ray type, i.e., rays are traced with R_{offset} = 0 and R_{stride} = 1. The scene geometries are colored by the hit group record that will be called when rays intersect the geometry. The bunny uses the first hit group record by specifying $\mathbb{I}_{offset}$ = 0; the instance of the teapot and Lucy specifies $\mathbb{I}_{offset}$ = 1 to use the hit group records following the bunny's. The teapot and Lucy geometries each have their own hit group record, which are accessed based on their $\mathbb{G}_{ID}$.*

different naming convention: R_{offset} is named I_{ray}, and R_{stride} is named $\mathbb{G}_{\text{mult}}$. Here, they are renamed to indicate that these parameters are specified when calling `TraceRay`.

The equation used to determine which hit group record is called when a ray hits a geometry in the scene is

$$\text{HG}_{\text{index}} = \mathbb{I}_{\text{offset}} + R_{\text{offset}} + R_{\text{stride}} \times \mathbb{G}_{\text{ID}}, \tag{15.1}$$

$$\text{HG} = \text{addByteOffset}(\&\text{HG}[0], \text{HG}_{\text{stride}} \times \text{HG}_{\text{index}}), \tag{15.2}$$

where &HG[0] is the starting address of the array containing the hit group records and $\text{HG}_{\text{stride}}$ is the stride in bytes between hit group records in the array. It is important to note that $\mathbb{I}_{\text{offset}}$ is not multiplied with R_{stride} in Equation (15.1). When using multiple ray types, i.e., $R_{\text{stride}} > 1$, this factor must typically be accounted for when computing each instance's $\mathbb{I}_{\text{offset}}$. Figure 15-2 illustrates the hit groups associated with each instance's geometry in a small scene for a ray tracer with a single ray type.

15.2.2 MISS RECORDS

The indexing rules for miss records are much simpler than those used for hit group records. The `TraceRay` call takes an index into the miss record table, R_{miss}, which specifies the index of the miss record to call. Though many ray tracers use a different miss record per ray type, and thus $R_{\text{offset}} = R_{\text{miss}}$, this is not required as the two parameters are specified independently. The miss record location is computed by

$$\mathrm{M}_{\text{index}} = R_{\text{miss}}, \tag{15.3}$$

$$\mathrm{M} = \text{addByteOffset}(\&\mathrm{M}[0], \mathrm{M}_{\text{stride}} \times \mathrm{M}_{\text{index}}), \tag{15.4}$$

where &M[0] is the starting address of the table containing the miss records and $\mathrm{M}_{\text{stride}}$ is the stride between miss records in bytes.

15.3 API-SPECIFIC DETAILS

Now that we have a unified terminology to work with when discussing the SBT and the parameters that affect its indexing, we can look into how these parameters are specified in each GPU ray tracing API. The biggest difference between the APIs, beyond terminology, is in how the embedded parameters are passed in the SBT and what types of parameters can be embedded. The $\mathbb{G}_{\text{ID}}$ parameter is set in the same way across the APIs, as it is just the index of the individual geometry in the BLAS's array of geometries.

15.3.1 DIRECTX RAYTRACING

The focus of this chapter is just on the SBT and aspects of the API related to its indexing; for more information about DXR, see the DXR chapter in *Ray Tracing Gems* [12], the DXR course at SIGGRAPH [11], the Microsoft Developer Network DXR documentation [4], the DXR HLSL Documentation [5], and the DXR specification [6].

EMBEDDED SHADER RECORD PARAMETERS

DXR supports embedding 8-byte handles to GPU objects, such as buffers and textures, or pairs of 4-byte constants in the SBT. Single 4-byte constants can also be embedded in the SBT, but must be padded to 8 bytes. When embedding multiple 4-byte constants, the constants should be paired together to avoid introducing unnecessary additional padding. The mapping of the embedded parameters in the shader record to the HLSL shader registers is specified using a *local root signature*.

INSTANCE PARAMETERS

Instances in DXR are specified through the following structure.

```
typedef struct D3D12_RAYTRACING_INSTANCE_DESC {
    FLOAT Transform[3][4];
    UINT InstanceID                          : 24;
    UINT InstanceMask                        : 8;
    UINT InstanceContributionToHitGroupIndex : 24;
    UINT Flags                               : 8;
    D3D12_GPU_VIRTUAL_ADDRESS AccelerationStructure;
} D3D12_RAYTRACING_INSTANCE_DESC;
```

Here are the parameters that affect SBT indexing:

- `InstanceContributionToHitGroupIndex`: The instance's SBT offset, $\mathbb{I}_{\text{offset}}$.
- `InstanceMask`: Though the mask does not affect which hit group is called, it can be used to skip traversal of instances entirely by masking them out.

TRACE RAY PARAMETERS

The HLSL `TraceRay` function can be called in the ray generation, closest-hit, and miss shaders to trace rays through the scene. `TraceRay` takes the acceleration structure to trace against, a set of ray flags to configure the traversal being performed, the instance mask, parameters that affect SBT indexing for the ray, the ray itself, and the payload to be updated by the closest-hit, any-hit, or miss shaders.

```
Template<payload_t>
void TraceRay(RaytracingAccelerationStructure AccelerationStructure,
              uint RayFlags,
              uint InstanceInclusionMask,
              uint RayContributionToHitGroupIndex,
              uint MultiplierForGeometryContributionToHitGroupIndex,
              uint MissShaderIndex,
              RayDesc Ray,
              inout payload_t Payload);
```

Here are the parameters that affect SBT indexing:

- `InstanceInclusionMask`: The mask affects which instances are traversed by taking the bitwise `AND` of the mask and each instance's `InstanceMask`. If this is zero, the instance is ignored.
- `RayContributionToHitGroupIndex`: The ray's SBT offset, R_{offset}.

- `MultiplierForGeometryContributionToHitGroupIndex`: The ray's SBT stride, R_{stride}.
- `MissShaderIndex`: The index of the miss shader record to call, R_{miss}.

15.3.2 VULKAN KHR RAY TRACING

For more documentation about the Vulkan KHR Ray Tracing extension, see Chapter 16, the extension specifications [2, 3], and the `GLSL_EXT_Ray_Tracing` extension [1].

SHADER RECORDS AND PARAMETERS

Vulkan treats the embedded parameters as a buffer block and supports most of the types of parameters that can be passed through a regular storage buffer (e.g., runtime sized arrays cannot be passed through the shader record). The embedded parameters are accessed in the shader through a buffer type declared with the `shaderRecordEXT` layout. Buffer handles can also be passed through the SBT in Vulkan by using the GLSL extension `GLSL_EXT_buffer_reference2`. The address of a buffer can be retrieved using `vkGetBufferDeviceAddress` and written to the SBT as an 8-byte constant. The following closest-hit shader demonstrates this approach to pass references to the index and vertex buffers of the geometry directly in the SBT. The example also uses the extension `GLSL_EXT_scalar_block_layout` to allow using a C-like structure memory layout in buffers.

```
layout(buffer_reference, buffer_reference_align=8, scalar)
buffer VertexBuffer {
    vec3 v[];
};

layout(buffer_reference, buffer_reference_align=8, scalar)
buffer IndexBuffer {
    uvec3 i[];
};

layout(shaderRecordEXT, std430) buffer SBT {
    VertexBuffer vertices;
    IndexBuffer  indices;
};

void main() {
    const uvec3 idx = indices.i[gl_PrimitiveID];
    const vec3 va = vertices.v[idx.x];
    const vec3 vb = vertices.v[idx.y];
    const vec3 vc = vertices.v[idx.z];
    // ...
}
```

INSTANCE PARAMETERS

The memory layout of Vulkan's instance structure matches that of DXR:

```
struct VkAccelerationStructureInstanceKHR {
    VkTransformMatrixKHR transform;
    uint32_t instanceCustomIndex                    : 24;
    uint32_t mask                                   : 8;
    uint32_t instanceShaderBindingTableRecordOffset : 24;
    VkGeometryInstanceFlagsKHR flags                : 8;
    uint64_t accelerationStructureReference;
};
```

Here are the parameters that affect SBT indexing:

- `instanceShaderBindingTableRecordOffset`: The instance's SBT offset, $\mathbb{I}_{\text{offset}}$.
- `mask`: Though the mask does not affect which hit group is called, it can be used to skip traversal of instances entirely by masking them out.

TRACE RAY PARAMETERS

The GLSL `traceRayEXT` function from `GLSL_EXT_ray_tracing` can be called in the ray generation, closest-hit, and miss shaders to trace rays through the scene. The function takes the acceleration structure to trace against, a set of ray flags to adjust the traversal being performed, the instance mask, parameters that affect SBT indexing for the ray, the ray itself, and the index of the ray payload to be updated by the closest-hit, any-hit, or miss shaders.

```
void traceRayEXT(accelerationStructureEXT topLevel,
                 uint rayFlags,
                 uint cullMask,
                 uint sbtRecordOffset,
                 uint sbtRecordStride,
                 uint missIndex,
                 vec3 origin,
                 float Tmin,
                 vec3 direction,
                 float Tmax,
                 int payload);
```

Here are the parameters that affect SBT indexing:

- `cullMask`: The mask affects which instances are traversed by taking the bitwise `AND` of the mask and each instance's `mask`. If this is zero, the instance is ignored.
- `sbtRecordOffset`: The ray's SBT offset, R_{offset}.

- `sbtRecordStride`: The ray's SBT stride, R_{stride}.
- `missIndex`: The index of the miss shader to call, R_{miss}.

In contrast to HLSL, ray payloads are specified as shader input/output variables, where the value of `payload` selects which will be used. For example:

```
struct RayPayload {
    vec3 hit_pos;
    vec3 normal;
};
layout(location = 0) rayPayloadEXT RayPayload payload;
```

15.3.3 OPTIX

For more documentation about OptiX [8], see the OptiX 7 programming guide [7] and the OptiX 7 course at SIGGRAPH [10].

SHADER RECORDS AND PARAMETERS

OptiX supports embedding arbitrary structs in the SBT, and these structs can contain CUDA device pointers to buffers or texture handles. A pointer to the embedded parameters for the shader record being called can be retrieved by calling `optixGetSbtDataPointer`. This function returns a `void*` to the portion of the SBT after the shader handle containing the embedded parameters.

INSTANCE PARAMETERS

Instances in OptiX are specified through the `OptixInstance` structure:

```
struct OptixInstance {
    float transform[12];
    unsigned int instanceId;
    unsigned int sbtOffset;
    unsigned int visibilityMask;
    unsigned int flags;
    OptixTraversableHandle traversableHandle;
    unsigned int pad[2];
};
```

Here are the parameters that affect SBT indexing:

- `sbtOffset`: The instance's SBT offset, $\mathbb{I}_{\text{offset}}$.
- `visibilityMask`: Though the mask does not affect which hit group is called, it can be used to skip traversal of instances entirely by masking them out.

TRACE RAY PARAMETERS

The `optixTrace` function can be called in the ray generation, closest-hit, and miss shaders to trace rays through the scene. The function takes the acceleration structure to trace against, a set of ray flags to adjust the traversal being performed, the instance mask, parameters that affect SBT indexing for the ray, the ray itself, and up to 8 unsigned 4-byte values that are passed by reference through registers and can be modified in the closest-hit, any-hit, or miss shader. To pass a struct larger than 32 bytes (8×4), it is possible to pass a pointer to a stack variable in the calling shader by splitting the pointer into two 32-bit `int`s and reassembling the pointer later in the shader.

```
void optixTrace(OptixTraversableHandle handle,
                float3 rayOrigin,
                float3 rayDirection,
                float tmin,
                float tmax,
                float rayTime,
                OptixVisibilityMask visibilityMask,
                unsigned int rayFlags,
                unsigned int SBToffset,
                unsigned int SBTstride,
                unsigned int missSBTIndex,
                // Up to 8 4-byte values to be passed
                // through registers.
                // Unsigned int& p0-p7
)
```

Here are the parameters that affect SBT indexing:

- `visibilityMask`: The mask affects which instances are traversed by taking the bitwise `AND` of the mask and each instance's `visibilityMask`. If this is zero, the instance is ignored.
- `SBToffset`: The ray's SBT offset, R_{offset}.
- `SBTstride`: The ray's SBT stride, R_{stride}.
- `missSBTIndex`: The miss shader index to call, R_{miss}.

15.4 COMMON SHADER BINDING TABLE CONFIGURATIONS

Now that we have a unified terminology to discuss the SBT, the scene parameters, and the trace ray parameters and have seen how this terminology maps to the specific GPU ray tracing APIs, we can discuss a few common ray tracer configurations to provide guidance through hands-on examples. The examples are discussed in an API-agnostic manner, and hit group records will be shown tightly packed together in the SBT; however, in

practice some padding may be needed between hit group records, depending on the API being used and the embedded parameters being passed.

15.4.1 A BASIC RAY TRACER

The basic ray tracer example uses two ray types, and each geometry has a hit group per ray type, i.e., $R_{\text{stride}} = 2$. The first ray type, $R_{\text{offset}} = 0$, is used for primary rays, the second, $R_{\text{offset}} = 1$, is used for shadow rays. The primary hit group's closest-hit shader will compute and return the surface normal and material ID through the ray payload; the shadow hit group will mark the ray as occluded in the payload. Both hit groups will use the same any-hit shader, which will filter out intersections against parts of the geometry made transparent by an alpha cutout texture. A miss record is also provided for each ray type; the primary ray miss record returns a background color through the payload, while the shadow ray miss record marks the ray as unoccluded. The hit group records for this renderer are shown in Figure 15-3 on a scene with two instances. Each instance in this example has a unique BLAS, and thus a unique set of geometries.

Hit group records are written in order by instance and the geometry order within the instance's BLAS. Instances in the scene do not share hit groups,

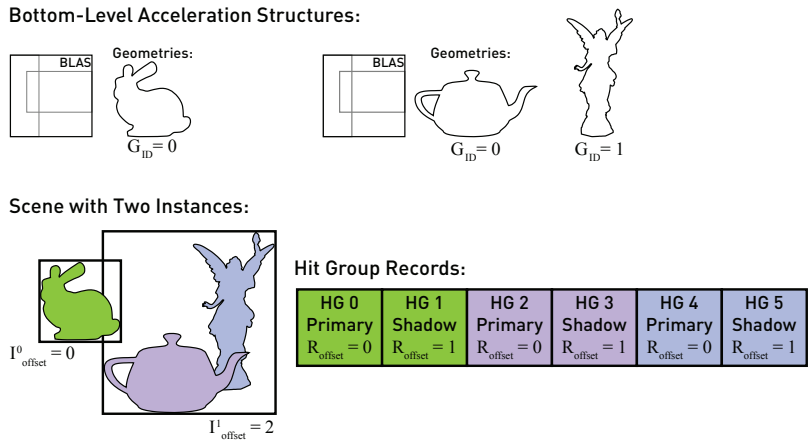


Figure 15-3. *The hit group records for a basic ray tracer with two ray types rendering a scene with two instances. One of the instance's BLAS has two geometries. Hit group records are color coded by the geometry in the scene by which they are used. Rays are traced with $R_{stride} = 2$, primary rays are traced with $R_{offset} = 0$, and shadow rays are traced with $R_{offset} = 1$.*

and thus the SBT offset for instance i is computed as

$$\mathbb{I}^i_{\text{offset}} = \mathbb{I}^{i-1}_{\text{offset}} + \mathbb{I}^{i-1}_{\text{geom}} \times R_{\text{stride}}, \tag{15.5}$$

$$\text{where } \mathbb{I}^0_{\text{offset}} = 0 \tag{15.6}$$

and where $\mathbb{I}^i_{\text{geom}}$ is the number of geometries in the BLAS used by instance i.

15.4.2 INSTANCING A BLAS WITH THE SAME HIT GROUP PARAMETERS

A common use case of instancing is to place the same mesh, i.e., BLAS, with the same material parameters, i.e., embedded shader parameters, at multiple locations in the scene. This can be done efficiently without increasing the size of the SBT by having each such instance share a single hit group record. Instead of assigning each instance a unique offset as done in the previous example, each unique instanced object is assigned an offset in the SBT. Instances of the same object then use the same $\mathbb{I}_{\text{offset}}$ to reference the same hit group records. Instance offsets must now be computed in two steps: First, we find the unique instanced objects in the scene to build a list of unique instance offsets:

$$\mathbb{UI}^i_{\text{offset}} = \mathbb{UI}^{i-1}_{\text{offset}} + \mathbb{UI}^{i-1}_{\text{geom}} \times R_{\text{stride}}, \tag{15.7}$$

$$\text{where } \mathbb{UI}^0_{\text{offset}} = 0. \tag{15.8}$$

The hit group records for each unique instanced object i are written in the SBT starting at its unique offset, $\mathbb{UI}^i_{\text{offset}}$. Then, the instances of this object can use $\mathbb{UI}^i_{\text{offset}}$ as their instance offset to share this single set of hit group records.

Figure 15-4 illustrates this configuration on a scene with three instances in a ray tracer with two ray types. Two of these instances reference the same unique instanced object, the green bunny. These two instances share the same unique instance offset:

$$\mathbb{I}^0_{\text{offset}} = \mathbb{I}^1_{\text{offset}} = \mathbb{UI}^0_{\text{offset}} = 0. \tag{15.9}$$

The next unique instance, with the teapot and Lucy, follows the hit group records shared by the bunny's, at $\mathbb{I}^2_{\text{offset}} = \mathbb{UI}^1_{\text{offset}} = 2$.

15.4.3 DROPPING THE SHADOW HIT GROUP WHEN RENDERING OPAQUE GEOMETRIES

A trick we can use to simplify SBT setup and reduce its size when rendering only opaque geometry is to use the ray flags to skip calling the any-hit and closest-hit shaders for shadow rays. Instead of using the closest-hit shader in

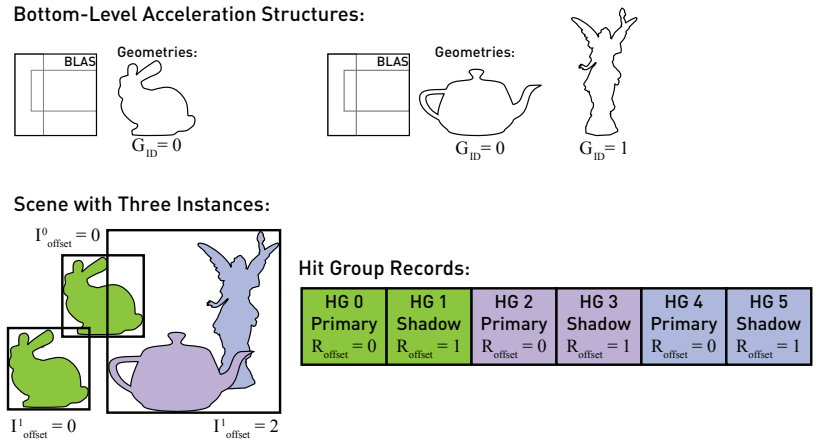


Figure 15-4. *The hit group records for a basic ray tracer with two ray types and instance hit group record reuse. The instances of the bunny place the same unique object in the scene and can thus reference the same hit group records by specifying the same $\mathbb{I}_{offset}$. Hit group records are color coded by the geometry in the scene that they are used by. Rays are traced with R_{stride} = 2, primary rays are traced with R_{offset} = 0 and shadow rays traced with R_{offset} = 1.*

the shadow hit group to mark rays as occluded, we will trace rays *assuming* they are occluded and use the miss shader to mark them unoccluded. The shadow hit group is no longer called, and it can be omitted from the SBT. As each geometry only has a single hit group, the R_{offset}, R_{stride}, and R_{miss} parameters will diverge from each other. Both primary and shadow rays are now traced with $R_{\text{offset}} = 0$ and $R_{\text{stride}} = 1$, while $R_{\text{miss}} = 0$ for primary rays and $R_{\text{miss}} = 1$ for shadow rays. This means that shadow rays reference the primary ray hit group. However, shadow rays are traced with ray flags specifying to skip calling the closest-hit and any-hit shaders and thus will not actually call this hit group. The only shader called when traversing shadow rays will be the miss shader, if the ray did not hit anything. Occluded shadow rays are processed entirely in hardware without any shaders run during traversal when using built-in hardware accelerated primitives (e.g., triangles). Figure 15-5 illustrates the resulting hit group record layout for this configuration.

This approach also works for custom geometries, as shadow rays will simply use the intersection shader provided in the primary ray hit group record. However, if it is possible to perform a faster intersection test for shadow rays that only checks if a hit exists instead of finding the closest one, it may be best to still provide a separate hit group for shadow rays.

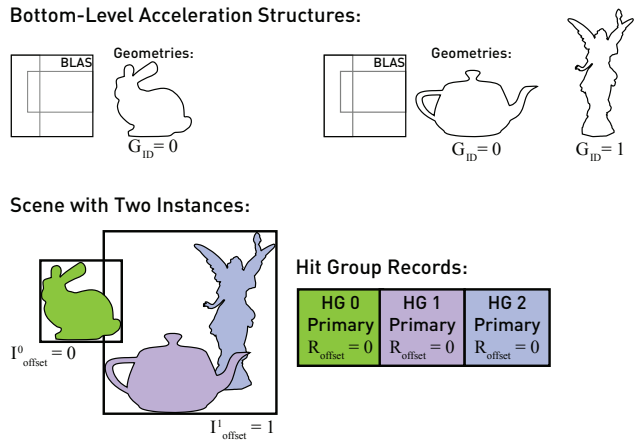


Figure 15-5. *The hit group records for a ray tracer without shadow hit groups. Both primary and shadow rays are traced with R_{offset} = 0 and R_{stride} = 1, with the ray flags for shadow rays set to skip the closest-hit and any-hit shaders. For primary rays R_{miss} = 0, while for shadow rays R_{miss} = 1. In this configuration, shadow rays are assumed to be occluded and are marked unoccluded by the miss shader.*

15.4.4 A MINIMAL ONE OR TWO HIT GROUP RAY TRACER

If all the geometry in the scene use the same hit group shaders, it is also possible to use a single hit group that is shared by all geometries to create a minimal SBT. This requires that all geometry in the scene use the same closest-hit shader and, in the case that the intersection or any-hit shaders are used, requires that all geometry again use the same ones. Rather than the parameters being sent for each geometry embedded in the hit group record, they are stored in globally accessible memory and retrieved using the instance, geometry, and primitive IDs. All instances then specify their SBT offset as $\mathbb{I}^i_{\text{offset}} = 0$. Note that the instance ID is specified separately from the SBT offset, allowing each instance to still specify a different ID to be used to look up the correct data for its geometry. Rays are traced with $R_{\text{stride}} = 0$ to cancel out the additional offset applied by $\mathbb{G}_{\text{ID}}$ within a BLAS. Figure 15-6 illustrates the hit group record configuration for a ray tracer with separate primary and shadow hit groups. This approach can be combined with that discussed in Section 15.4.3 when rendering opaque geometries, requiring only a single hit group in the SBT.

As this approach does not embed parameters in the SBT, it can require additional bandwidth to retrieve data from global memory, though this may

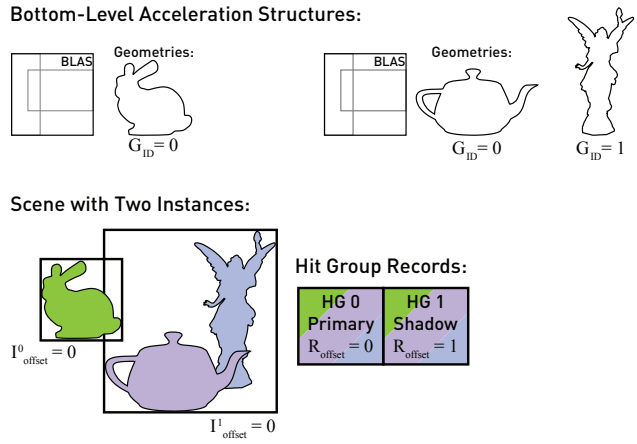


Figure 15-6. *The hit group records for a ray tracer where all geometries use the same primary and shadow hit groups. All rays are traced with $R_{stride} = 0$ to cancel out the $\mathbb{G}_{ID}$ offset that would otherwise be applied when looking up the hit group. Shadow rays are still traced with $R_{offset} = 1$ to access the shadow hit group. All object parameters are stored in globally accessible buffers that are indexed using the instance, geometry, and primitive IDs.*

simplify the process of bringing ray tracing pipelines into an existing bindless rasterizer that already stores geometry data in globally accessible buffers.

15.4.5 DYNAMICALLY UPDATING THE SBT

The examples so far have focused on static scenes, where the entire SBT is populated and uploaded to the GPU once at the start of the program. However, in real-time applications or when rendering dynamic scenes, reallocating, populating, and uploading the entire SBT each time the scene changes could consume an undesirable amount of bandwidth and impact performance. This is especially a concern in real-time applications such as games where the scene changes frequently. The SBT itself is just a buffer on the GPU that is used as a lookup table for each geometry's hit group records. There is no requirement that hit group records in this buffer are tightly packed nor that they fill the entire buffer's memory.

Instead, we can implement a linear allocator to assign memory to an instance's hit group records as instances enter and exit the scene. Similar to other strategies for linear allocators, we allocate a large buffer to hold the hit group records and make sub-allocations within this buffer. Instances that enter the scene request an allocation of $\mathbb{I}^i_{\text{geom}} \times R_{\text{stride}}$ hit group records from

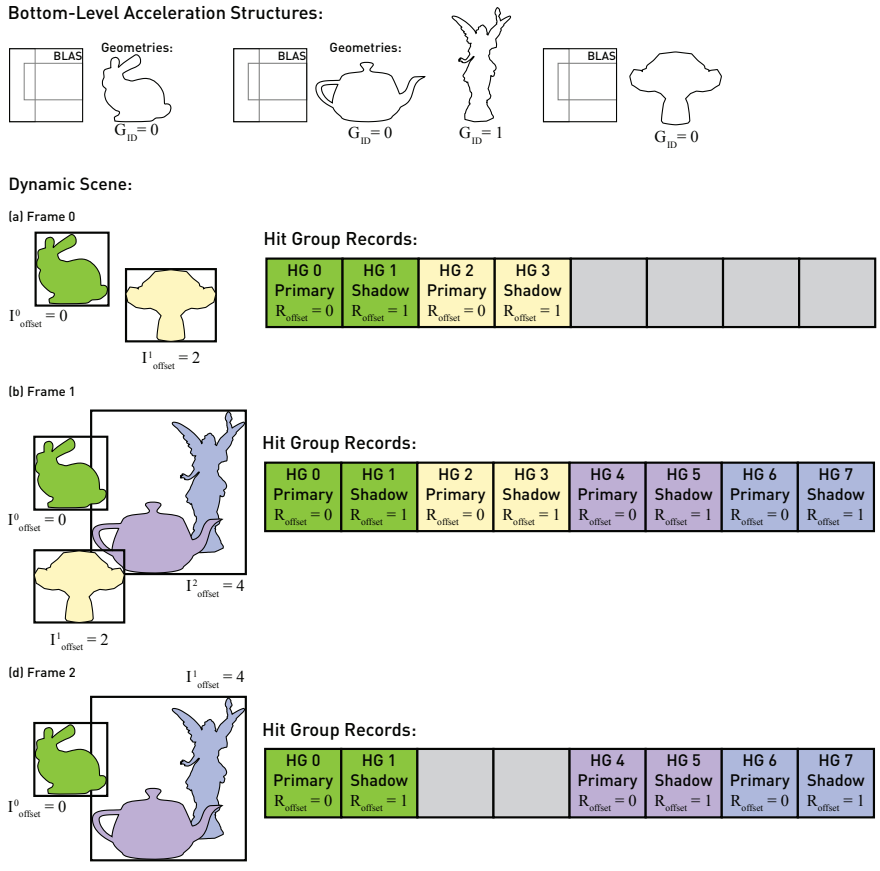


Figure 15-7. *Hit group records in the SBT can be managed using a linear allocator to handle dynamic scenes without having to reallocate and repopulate the entire SBT each time the scene changes. This example shows a ray tracer with two ray types using this approach to update the SBT as objects enter and exit the scene. (a) Frame 0 contains just the bunny and monkey, which partially fill the hit group records buffer. (b) The instance with the teapot and Lucy enters the scene in frame 1, taking the last four entries of the buffer. (c) The monkey exits the scene in frame 2, leaving an unused region of memory in the buffer.*

the allocator to write their hit group records into the SBT. The start of this allocation is the instance's offset, $\mathbb{I}^i_{\text{offset}}$. Each hit group record must still have the same stride, $\text{HG}_{\text{stride}}$, and each shader that may be used in the scene must have been included in the ray tracing pipeline when it was created. Figure 15-7 illustrates this approach on a dynamic scene over three frames as objects enter and exit the scene.

15.5 SUMMARY

This chapter has presented a holistic model of the SBT across the GPU ray tracing APIs to introduce the model in a consistent manner to developers new to these APIs. This model, combined with the hands-on examples, aims to provide a fundamental understanding of the SBT programming model. Similarly, this chapter aims to help developers familiar with some of these APIs translate their knowledge to others more easily through a unified SBT terminology and the API-specific mappings discussed. For developers new to GPU ray tracing, my recommendation is to try implementing some of these examples in your own renderer to get hands-on experience, starting from the basic example discussed in Section 15.4.1. Though the discussion in this chapter has not gone into API-specific details about memory alignment requirements and strides, padding required between hit group records, or how embedding parameters may affect strides and padding, the interactive SBT builder tool on my website [9] can be used to explore these aspects. The online tool lets you create your SBT, add parameters to hit group records, set up the scene being rendered, and virtually trace rays to see which hit group records are called for each geometry.

REFERENCES

[1] Khronos. GLSL ray tracing extension specification. https://github.com/KhronosGroup/GLSL/blob/master/extensions/ext/GLSL_EXT_ray_tracing.txt, 2021. Accessed February 3, 2021.

[2] Khronos. Vulkan acceleration structure specification. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_acceleration_structure.html, 2021. Accessed February 3, 2021.

[3] Khronos. Vulkan raytracing pipeline specification. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_ray_tracing_pipeline.html, 2021. Accessed February 3, 2021.

[4] Microsoft. Direct3D 12 Raytracing documentation. https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-raytracing, 2021. Accessed February 3, 2021.

[5] Microsoft. Direct3D 12 Raytracing HLSL shaders. https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-raytracing-hlsl-shaders, 2021. Accessed February 3, 2021.

[6] Microsoft. DirectX Raytracing (DXR) functional spec. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html, 2021. Accessed February 3, 2021.

[7] NVIDIA. OptiX 7 programming guide. https://raytracing-docs.nvidia.com/optix7/guide/index.html, 2021. Accessed February 3, 2021.

[8] Parker, S. G., Robison, A., Stich, M., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., and Morley, K. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics*, 29(4):66:1–66:13, 2010. DOI: 10.1145/1778765.1778803.

[9] Usher, W. The RTX shader binding table three ways. https://www.willusher.io/graphics/2019/11/20/the-sbt-three-ways, November 20, 2019.

[10] Wald, I. and Parker, S. G. OptiX 7 course. Course at SIGGRAPH, https://gitlab.com/ingowald/optix7course, 2019. Accessed February 3, 2021.

[11] Wyman, C., Hargreaves, S., Shirley, P., and Barré-Brisebois, C. Introduction to DirectX Raytracing. ACM SIGGRAPH Course, http://intro-to-dxr.cwyman.org, 2018.

[12] Wyman, C. and Marrs, A. Introduction to DirectX Raytracing. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 3. Apress, 2019.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 16

INTRODUCTION TO VULKAN RAY TRACING

Matthew Rusch, Neil Bickford, and Nuno Subtil
NVIDIA

ABSTRACT

Modern graphics APIs offer performant, low-level access to accelerated ray tracing. With this control and flexibility come complexity and nuance that can present challenges that are difficult for developers to navigate. This chapter will serve as an introduction to one method of harnessing the power of hardware-accelerated ray tracing: Vulkan and the Vulkan Ray Tracing extensions.

16.1 INTRODUCTION

Readers of the first *Ray Tracing Gems* book might recall that it contained a chapter with a similar title, "Introduction to DirectX Raytracing" [15]. This chapter is very similar in spirit and structure. For those who have not read the



Figure 16-1. *Vulkan Ray Tracing in action in Crysis Remastered.*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_16

aforementioned text, or anyone who would like a review, a brief introduction of ray tracing in the context of the Vulkan API follows.

Consumer-grade ray tracing acceleration hardware was released in 2018. Varying methods of programmable hardware access were announced shortly after. One of these ways of controlling hardware acceleration is via Vulkan API extensions. In late 2020 the official cross-platform Vulkan Ray Tracing (VKR) extensions were finalized and released.

This chapter assumes that the reader has a working knowledge of the core Vulkan API or, at a minimum, experience with a modern low-level graphics API such as Direct3D 12 or console graphics APIs for instance. Additionally, a fundamental understanding of ray tracing is required. See the first *Ray Tracing Gems* volume or an introductory text if necessary.

This chapter focuses on ray tracing pipelines and acceleration structures; however, the Vulkan Ray Tracing extensions also include `VK_KHR_ray_query` for ray queries. These permit ray tracing in non–ray tracing shader stages such as fragment and compute stages, but without the structure of ray tracing shaders, much like inline ray tracing in DirectX Raytracing (DXR) Tier 1.1. Ray queries will not be discussed here, but information regarding them may be found in the supplementary material listed at the end of this chapter.

16.2 OVERVIEW

Modern graphics programming, which utilizes the GPU, has been a cross-disciplinary endeavor for some time. It requires mathematical fundamentals, such as linear algebra and Cartesian geometry. It also requires knowledge of CPU-side API constructs in order to coordinate rendering tasks. Additionally, familiarity with GPU-side operations, such as shaders and fixed-function operations, is necessary to do the actual rendering work. This chapter will introduce the requisite parts of each of these and show them applied contextually using a sample ray tracing application, which is included in the code for this book.

Real-time ray tracing, in the context of modern graphics APIs, can be broken down into three main high-level concepts. The first of these is the *ray tracing pipeline*, which contains details about execution and state. The ray tracing pipeline contains the shaders that are executed during a ray dispatch and also describes how they exchange data. The second concept is the use of *acceleration structures* (AS), which are spatial data structures that organize

and abstract scene geometry and enable more efficient scene traversal. They serve to accelerate ray tracing and intersection testing. The final concept is the *shader binding table* (SBT), which is essentially a large array of shader references with interleaved per-object data. It defines the relationship between the ray tracing shaders (within the pipeline), their respective resources, and the scene geometry.

Section 16.3 will prepare the reader to set up their own Vulkan Ray Tracing application with respect to both software and hardware requirements. Sections 16.4 through 16.6 walk through core shading concepts and components, and introduce how to write ray tracing shaders.

Sections 16.7 and 16.8 cover the required CPU-side setup to initialize Vulkan and create the objects necessary for ray tracing, including acceleration structure building and shader compilation. These concepts are applied in Sections 16.9 and 16.10, where they will be used to populate the ray tracing pipeline and shader binding table, two constructs that dictate how data is related and operated on between CPU and GPU. This culminates in Section 16.11, which shows how to bind the necessary pipeline objects and dispatch rays.

16.3 GETTING STARTED

The amount of system setup to get a ray tracing application up and running is minimal. Some of this is platform dependent; a platform that supports Vulkan is required, as is a means of compiling code that uses Vulkan. Vulkan SDK 1.2.162.0 and later now fully support the Vulkan Ray Tracing extensions, including validation and toolchain support. This is available for download at the LunarG Vulkan SDK website [8].

Hardware and drivers that support Vulkan and the ray tracing extensions are also required and are available from multiple vendors. Additionally, there is a wealth of samples and publications available on the Internet that can be used to augment the material contained in this chapter. Please see the "Additional Resources" (Section 16.12) and "References" sections for pointers on where to go next.

16.4 THE VULKAN RAY TRACING PIPELINE

The GPU ray tracing pipeline (as shown in Figure 16-2) looks much different than its traditional rasterization-based graphics counterpart. It is similar in

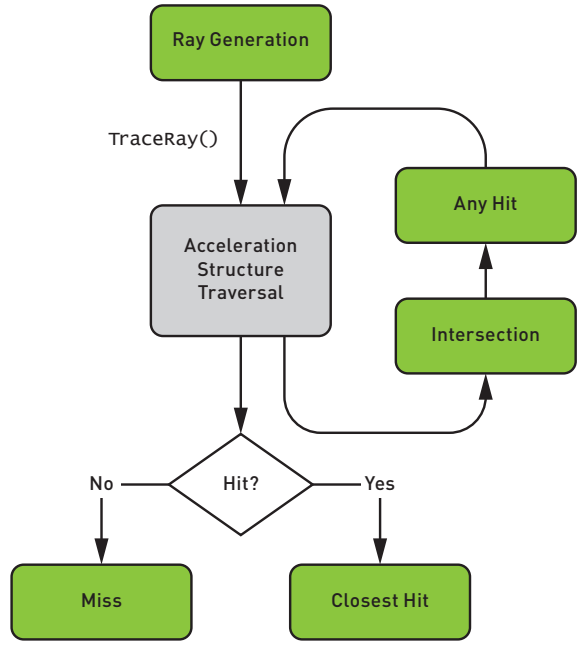


Figure 16-2. *Graph of the Vulkan Ray Tracing pipeline, showing shader stage interactions.*

that there are multiple, separate, programmable stages intended for use by developers to control rendering. The ray tracing pipeline, however, uses a different set of shaders that interact with each other in a different manner than the shaders in the rasterization pipeline. It also introduces new fundamental concepts, such as the ray and ray payload, which are instrumental in shader operation and facilitate communication between shader stages. The `VK_KHR_ray_tracing_pipeline` extension provides the functionality of the ray tracing pipeline and shader stages.

An enumeration and brief introduction of the ray tracing shader stages follows:

- Ray generation shader.
- Miss shader.
- Closest-hit shader.
- Intersection shader.
- Any-hit shader.

Ray generation shaders are the starting point for the ray tracing pipeline. They are similar to compute shaders, except with the added ability of being able to specify and launch rays with the HLSL `TraceRay()` function (`traceRayEXT()` in GLSL), and they are typically launched in a regular grid of indices in up to three dimensions. The `TraceRay()` function casts a single ray into the scene to test for intersections and, in doing so, may trigger other shaders, which will be introduced shortly, in the process. These other shaders are responsible for further processing and will return their results to the ray generation shader.

A *miss shader* is executed when a given ray does not intersect with any geometry. It may sample from an environment map, for example, or simply return a color. This shader and the closest-hit shader are capable of recursively spawning rays by calling `TraceRay()` themselves.

Closest-hit shaders are one of the three types of shaders in hit groups and are invoked on the closest-hit point along a ray. Vulkan ray tracing pipelines can use more than one closest-hit shader; which one is used for a particular intersection is governed by the pipeline's shader binding table and the traced acceleration structure, with details described later. This shader is where applications typically perform lighting calculations, material evaluation, etc. Closest-hit shaders are arguably the closest analog the ray tracing pipeline has to fragment shaders.

The final two shaders, which are introduced next, are optional.

Intersection shaders are used to intersect rays with user-defined geometry. Ray/triangle intersections have built-in support and thus do not require an intersection shader.

Ray traversal through the scene may result in intersections with multiple primitives. *Any-hit shaders* are executed on each intersection. They can be used to discard intersections, to enable alpha testing for example, by performing a texture lookup and ignoring the intersection if the fetched value does not meet a specified criteria. The default any-hit shader is a pass-through that returns information about the intersection to the traversal engine so that it can determine the closest intersection.

The programming model and pipeline just described consider a single ray at a time from the perspective of the programmer. Rays are isolated from other rays and do not interact. However, the various shader stages being executed while evaluating a ray's trajectory can communicate by using *ray payloads*. The ray payload is an arbitrary, user-defined structure passed as an `inout`

parameter to the `TraceRay()` function in HLSL, or via a `rayPayloadEXT`/`rayPayloadInEXT` variable in GLSL. The hit and miss shaders can modify this structure, enabling them to pass information between stages and eventually back to the ray generation function, which will use this information to write results to memory.

16.5 HLSL/GLSL SUPPORT

Vulkan Ray Tracing requires a number of extensions to GLSL and HLSL to expose the new functionality it provides. This new functionality is exposed to shaders when compiling to SPIR-V.

16.5.1 GLSL

GLSL shaders obtain access to ray tracing features by enabling the `GLSL_EXT_ray_tracing` extension [3]. New storage qualifiers are defined for the various bits of data that need to be shared between shaders.

`rayPayloadEXT` declares a variable with storage for a ray payload. This is typically a small struct that stores ray properties that are needed in hit shaders, and it is also used to pass hit information from hit shaders back to the shader tracing the ray. It is allowed on any shader stages that are able to call `TraceRay()`.

`rayPayloadInEXT` declares an input ray payload variable, without storage (which is expected to be allocated in a different stage via the `rayPayloadEXT` qualifier). This is allowed on any stage that can be invoked during execution of `TraceRay()`. The variable type used in `rayPayloadEXT` and `rayPayloadInEXT` must match between caller and callee.

`hitAttributeEXT` declares a variable with storage for ray/primitive intersection data. Explicitly declaring hit attribute storage is only required when the pipeline includes custom intersection shaders. Default triangle intersection shaders use an implicit `hitAttributeEXT` variable layout consisting of a `vec2` with the barycentric coordinates of the intersection point. The hit attribute variable is read-only for any-hit and closest-hit shaders and is only read-write in intersection shaders. Storage is implicitly allocated whenever this qualifier is used in an intersection shader.

The `GLSL_EXT_ray_tracing` extension also defines a number of built-in variables and functions to manipulate and trace rays through the scene. The most commonly used ones are covered here:

- `gl_LaunchIDEXT` and `gl_LaunchSizeEXT` identify a thread in the launch grid for a ray generation shader.
- `gl_PrimitiveIDEXT` and `gl_InstanceIDEXT` identify a primitive when processing an intersection in a closest-hit, any-hit, or intersection shader.
- `gl_HitTEXT` contains the *t*-value of an intersection along the ray (accessible in closest-hit and any-hit shaders).

Finally, a number of built-in functions are defined, the most widely used being `traceRayEXT()`, which initiates a ray traversal operation. This can be called from ray generation, closest-hit, and miss shaders.

The full specification for `GLSL_EXT_ray_tracing` is available from Khronos [3].

16.5.2 HLSL

Microsoft's DirectX Shader Compiler (DXC) includes support for generating SPIR-V bytecode from HLSL source shaders. This has been extended to include support for `KHR_ray_tracing` by mapping the equivalent DirectX Raytracing language primitives, thus allowing DXR shaders to be run on Vulkan.

SHADER STAGES

Unlike the case with pixel and vertex shader stages, there are no explicit new profile names for ray tracing shader stages. Rather, all ray tracing stages are compiled using the `lib_6_3` or `lib_6_4` target profiles.

Shader stages are identified by annotations on the entry point: `[shader("raygeneration")]` declares an entry point as the start of a ray generation shader, and similarly for intersection shaders (`[shader("intersection")]`), hit shaders (`[shader("closesthit")]` or `[shader("anyhit")]`), and miss shaders (`[shader("miss")]`).

INTRINSIC VARIABLES AND FUNCTIONS

A very similar set of intrinsic variables is declared in DXR as well as GLSL for use in ray tracing shaders, with similar semantics.

The DirectX Shader Compiler repository on GitHub [9] contains extensive documentation on the mappings between HLSL and SPIR-V.

SHADER RECORD BUFFER AND LOCAL ROOT SIGNATURES

DXR and `VK_KHR_ray_tracing_pipeline` differ in one significant way: per-object resource binding.

Both DXR and `VK_KHR_ray_tracing_pipeline` define a table of shader records, where data that is specific to a combination of a given geometry instance and shader stage can be stored.

In DXR, the shader record table can contain constant values, as well as resource binding information (known as a *local root signature*). However, Vulkan allows only constant values and has no primitive equivalent to the local root signature. Therefore, when porting to Vulkan, DXR shaders that make use of the local root signature for per-record resource binding will need to be modified to use only constant values in the shader record. Constant values in the shader record are exposed to HLSL using annotation `[[vk::shader_record_nv]]`, which can only be applied to constant buffer declarations. This annotation (along with any other annotations following the `[[vk::*]]` naming convention) is Vulkan-specific; however, it will be ignored when compiling for DirectX Intermediate Language (DXIL), so the same shader code can still be used across both Vulkan and DXR.

16.6 RAY TRACING SHADER EXAMPLE

For an example of how ray tracing shaders work, consider the following GLSL code samples that use ray tracing to compute ambient occlusion (AO), and see the generated result in Figure 16-3. This system uses one ray generation shader, one miss shader, and no ray tracing shaders of any other type.

The ray generation shader will be invoked once per pixel. Each invocation will first look up its pixel's corresponding world-space position and normal. To determine if a given direction is occluded, the invocation will initialize a ray payload, then trace a ray, making use of the fact that the miss shader is only invoked if the ray reports no intersections. When called, the miss shader will set a flag in the payload indicating that the direction was unoccluded. The ray generation shader can then read this payload to determine if the direction was occluded.

First, the structure for the ray payload is defined. This definition is done in a file, `ao_shared.h`, that is included by both the ray generation and miss shaders (Listing 16-1).



Figure 16-3. *Ray traced ambient occlusion generated using this chapter's sample code.*

Listing 16-1. *ao_shared.h.*

```
struct RayPayload // Define a ray payload.
{
  float hitSky;  // 0 = occluded, 1 = visible
};
```

The ray generation shader sets `hitSky` to 0, then calls `traceRayEXT()`. If called, the miss shader then accesses the ray payload with which it was called and sets `hitSky` to 1 (Listing 16-2).

Listing 16-2. *The GLSL miss shader in* `ao.rmiss`.

```
#version 460
#extension GL_EXT_ray_tracing : require
#extension GL_GOOGLE_include_directive : require
#include "ao_shared.h"

layout(location = 0) rayPayloadInEXT RayPayload pay;

void main()
{
  pay.hitSky = 1.0f;
}
```

The ray generation shader wraps `traceRayEXT()` in a function called `ShadowRay()`, which will return 1 if the given direction was occluded from the given origin and 0 otherwise. The ray generation shader also accesses the ray payload from GLSL location 0 (Listing 16-3).

Listing 16-3. *The payload and* `ShadowRay` *definition in* `ao.rgen`.

```
#version 460
#extension GL_EXT_ray_tracing : require
#extension GL_GOOGLE_include_directive : require
#include "ao_shared.h"

// A location for a ray payload (we can have multiple of these)
layout(location = 0) rayPayloadEXT RayPayload pay;

float ShadowRay(vec3 orig, vec3 dir)
{
  pay.hitSky = 0.0f;  // Assume ray is occluded
  traceRayEXT(
      scene,   // Acceleration structure
      // Ray flags, here saying "ignore any-hit shaders and
      // closest-hit shaders, and terminate the ray on the
      // first found intersection"
      gl_RayFlagsOpaqueEXT | gl_RayFlagsSkipClosestHitShaderEXT |
          gl_RayFlagsTerminateOnFirstHitEXT,
      0xFF,    // 8-bit instance mask
      0,       // SBT record offset
      0,       // SBT record stride for offset
      0,       // Miss index
      orig,    // Ray origin
      0.0,     // Minimum t-value
      dir,     // Ray direction
      1000.0,  // Maximum t-value
      0);      // Location of payload
  return pay.hitSky;
}
```

With this in place, ambient occlusion can be computed by calling `ShadowRay()` with random directions for each pixel and averaging the result, as shown in Listing 16-4.

The `GetPixelInfo()` function is a user-implemented function that returns a pixel's corresponding world-space position and normal, as well as whether it shows an object or the sky (in which case it does not compute ambient occlusion). This could be implemented by reading from a G-buffer or by tracing a ray into the scene (this chapter's sample code performs the latter). The `OffsetPositionAlongNormal()` function offsets the position a certain distance along the geometric normal to avoid self-intersection using the method of Wächter and Binder [14]. Each time `GetRandCosDir(norm)` is called, it returns a random direction in the hemisphere defined by `norm`, with the probability distribution cosine-weighted in the direction of `norm`. For the implementations of these functions, please see this chapter's sample code.

Listing 16-4. *Ambient occlusion by tracing rays in* `ao.rgen`.

```
layout(..., r8) uniform image2D imageAO;  // Output AO image
layout(...) uniform accelerationStructureEXT scene;  // Built AS

void main()
{
  // Determine this pixel's world-space position and normal,
  // whether by using ray tracing or using data from a G-buffer.
  uvec2 pixel = gl_LaunchIDEXT.xy;
  bool pixelIsSky; // Does the pixel show the sky (not a mesh)?
  vec3 pos, norm;  // AO rays from where?
  GetPixelInfo(pixel, pixelIsSky, pos, norm);
  if(pixelIsSky){
    // Don't compute ambient occlusion for the sky.
    imageStore(imageAO, ivec2(pixel), vec4(1.0));
    return;
  }

  // Avoid self-intersection.
  pos = OffsetPositionAlongNormal(pos, norm);

  // Compute ambient occlusion.
  float aoColor = 0.0;
  for(uint i = 0; i < 64; i++)  // Use 64 rays.
    aoColor += ShadowRay(pos, GetRandCosDir(norm)) / 64.0;
  imageStore(imageAO, ivec2(pixel), vec4(aoColor));
}
```

16.7 OVERVIEW OF HOST INITIALIZATION

Until now, the focus has been on what happens during ray traversal and how to write code that the GPU executes during traversal. However, the setup required to perform ray tracing in Vulkan has not yet been discussed. In order to ray trace a scene, an application must do the following:

1. Choose and initialize a Vulkan device that supports the appropriate ray tracing extensions.
2. Specify the scene geometry and build an acceleration structure.
3. Ensure that resources can be accessed from ray tracing stages.
4. Create a lookup table for the shaders in the scene, and use this to create a ray tracing pipeline and at least one shader binding table.
5. Dispatch a command to run the ray tracing pipeline with the shader binding tables.

Sections 16.8–16.10 describe how to initialize Vulkan Ray Tracing following these steps. When using an engine or framework supporting ray tracing, these steps may already be implemented.

Although the process of initialization can be long, much of the API can be motivated by the vast challenges that the Vulkan Ray Tracing API solves and the flexibility that it provides. For instance, a system might have multiple GPUs, necessitating finding and choosing one that supports ray tracing. Scenes may change over time and can include different vertex formats, instanced objects, and shader-defined geometry, so support for this appears in the API for building acceleration structures. Because rays are arbitrary and may intersect any nonempty geometry, the API uses tables of shader bindings. Shader binding tables are flexible enough to provide support for changing shaders between camera and shadow rays, for instance.

16.8 VULKAN RAY TRACING SETUP

Systems can have multiple GPUs, each of which may support a different set of Vulkan extensions. Because of this, a ray tracing application must select a physical device that supports the necessary Vulkan Ray Tracing extensions, then create a logical device with the physical device and extensions. Vulkan KHR Ray Tracing pipelines require using Vulkan API version at least 1.1, the `VK_KHR_ray_tracing_pipeline` extension, and the following extensions on which `VK_KHR_ray_tracing_pipeline` depends:

- `VK_KHR_acceleration_structure`.
- `VK_KHR_deferred_host_operations`.
- `VK_EXT_descriptor_indexing`.
- `VK_KHR_buffer_device_address`.
- `VK_KHR_get_physical_device_properties2`.
- `VK_KHR_shader_float_controls`.
- `VK_KHR_spirv_1_4`.

The last five extensions are part of Vulkan 1.2, so one option is to create a Vulkan 1.2 instance and a `VkDevice` with the `VK_KHR_ray_tracing_pipeline`, `VK_KHR_acceleration_structure`, and `VK_KHR_deferred_host_operations` extensions.

The sample code in Listing 16-5 finds and creates a `VkDevice` with the Vulkan Ray Tracing pipeline extensions, using Vulkan 1.2. After creating an instance with Vulkan 1.2, use `vkEnumeratePhysicalDevices()` to list all physical

Listing 16-5. *Finding a physical device supporting ray tracing pipelines.*

```
VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
std::vector<const char*> deviceExtensions =
{VK_KHR_ACCELERATION_STRUCTURE_EXTENSION_NAME,
  VK_KHR_DEFERRED_HOST_OPERATIONS_EXTENSION_NAME,
  VK_KHR_RAY_TRACING_PIPELINE_EXTENSION_NAME};

// Find a physical device that supports all above extensions.
for(VkPhysicalDevice consideredDevice : physicalDevices)
{
  // Get the list of extensions the device supports.
  uint32_t numExtensions;
  NVVK_CHECK(vkEnumerateDeviceExtensionProperties(
    consideredDevice, nullptr, &numExtensions, nullptr));

  std::vector<VkExtensionProperties>
    extensionProperties(numExtensions);
  NVVK_CHECK(vkEnumerateDeviceExtensionProperties(consideredDevice,
    nullptr, &numExtensions, extensionProperties.data()));

  if(AreExtensionsIncluded(deviceExtensions, extensionProperties))
  {
    physicalDevice = consideredDevice;
    break;
  }
}

assert(physicalDevice != VK_NULL_HANDLE);
```

devices on the system, then select a `VkPhysicalDevice` that supports `VK_KHR_acceleration_structure`, `VK_KHR_deferred_host_operations`, and `VK_KHR_ray_tracing_pipeline` using `vkEnumerateDeviceExtensionProperties()`.

In Listing 16-5, `NVVK_CHECK()` prints a message and throws an error if it was given a `VkResult` other than `VK_SUCCESS`, and `AreExtensionsIncluded()` returns whether all strings in its first argument are included in the vector of `VkExtensionProperties` in its second argument.

Next, in Listing 16-6, get the physical device ray tracing pipeline properties by passing a `VkPhysicalDeviceProperties2` structure extended with a `VkPhysicalDeviceRayTracingPipelinePropertiesKHR` structure to `vkGetPhysicalDeviceProperties2()`. This includes properties of SBTs used by Section 16.10.

Then, create a structure chain including `VkPhysicalDeviceFeatures2`, `VkPhysicalDeviceAccelerationStructureFeaturesKHR`, and `VkPhysicalDeviceRayTracingPipelineFeaturesKHR` objects, and query

Listing 16-6. *Querying physical device ray tracing properties.*

```
VkPhysicalDeviceProperties2 physicalDeviceProperties{
      VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2};
VkPhysicalDeviceRayTracingPipelinePropertiesKHR rtPipelineProperties{
      VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_PROPERTIES_KHR};
physicalDeviceProperties.pNext = &rtPipelineProperties;
vkGetPhysicalDeviceProperties2(physicalDevice, &physicalDeviceProperties);
```

which features are supported. Create the `VkDevice` using the list of extensions and using this structure chain as a list of features to enable, as in Listing 16-7.

Listing 16-7. *Querying features and creating the `VkDevice`.*

```
VkPhysicalDeviceFeatures2 features2{
      VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2};
VkPhysicalDeviceVulkan12Features features12{
      VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_2_FEATURES};
VkPhysicalDeviceVulkan11Features features11{
      VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_1_1_FEATURES};
VkPhysicalDeviceAccelerationStructureFeaturesKHR asFeatures{
      VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ACCELERATION_STRUCTURE_FEATURES_KHR};
VkPhysicalDeviceRayTracingPipelineFeaturesKHR rtPipelineFeatures{
      VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_FEATURES_KHR};

features2.pNext  = &features12;
features12.pNext = &features11;
features11.pNext = &asFeatures;
asFeatures.pNext = &rtPipelineFeatures;

// Query supported features.
vkGetPhysicalDeviceFeatures2(physicalDevice, &features2);

// Create the Vulkan device.
VkDeviceCreateInfo deviceCreateInfo{VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO};
deviceCreateInfo.enabledExtensionCount = uint32_t(deviceExtensions.size());
deviceCreateInfo.ppEnabledExtensionNames = deviceExtensions.data();
deviceCreateInfo.pEnabledFeatures     = nullptr;
deviceCreateInfo.pNext                = &features2;
deviceCreateInfo.queueCreateInfoCount = 1;
deviceCreateInfo.pQueueCreateInfos    = &queueInfo;
NVVK_CHECK(vkCreateDevice(
  physicalDevice, &deviceCreateInfo, nullptr, &device));
```

Finally, get addresses to each of the extensions' functions. For instance, this chapter's sample code loads `vkCmdTraceRaysKHR()` using `vkGetDeviceProcAddr()`, then defines a wrapper around calling this function pointer (Listing 16-8). The full list of Vulkan KHR Ray Tracing functions is omitted for brevity, but can be found at [12, `nvvk/extensions_vk.cpp`, Line 815].

Listing 16-8. *Loading and calling a function added by the ray tracing pipeline extension.*

```
PFN_vkCmdTraceRaysKHR pfn_vkCmdTraceRaysKHR
  = (PFN_vkCmdTraceRaysKHR)
    getDeviceProcAddr(device, "vkCmdTraceRaysKHR");

VKAPI_ATTR void VKAPI_CALL vkCmdTraceRaysKHR(
  VkCommandBuffer commandBuffer,
  const VkStridedDeviceAddressRegionKHR* pRaygenShaderBindingTable,
  const VkStridedDeviceAddressRegionKHR* pMissShaderBindingTable,
  const VkStridedDeviceAddressRegionKHR* pHitShaderBindingTable,
  const VkStridedDeviceAddressRegionKHR* pCallableShaderBindingTable,
  uint32_t width,
  uint32_t height,
  uint32_t depth)
{
  assert(pfn_vkCmdTraceRaysKHR);
  pfn_vkCmdTraceRaysKHR(
  commandBuffer,
  pRaygenShaderBindingTable,
  pMissShaderBindingTable,
  pHitShaderBindingTable,
  pCallableShaderBindingTable,
  width,
  height,
  depth);
}
```

16.8.1 ACCELERATION STRUCTURES

After `VkDevice` initialization, the next steps are to specify the geometry for the scene and to use this specification to build an acceleration structure for ray tracing. Ray tracing relies on finding ray/scene intersections. Imagine a mesh consisting of n triangles. To find a ray/mesh intersection, a naive algorithm might test the ray against every triangle in the mesh, taking $O(n)$ time. For large scenes, this would be prohibitively expensive. One way to make this faster is to use an acceleration structure, such as a bounding volume hierarchy (BVH), which groups triangles together into a tree of bounding boxes based on their position in the mesh (Figure 16-4). By using a tree structure, the number of potential intersections that must be considered for a given ray can be significantly reduced, for example from $O(n)$ to $O(\log n)$.

Because 3D scenes usually contain multiple objects, Vulkan uses a two-level acceleration structure format. This allows representing instances of objects with different transformations and material properties.

A *top-level acceleration structure* (TLAS) is an acceleration structure of *instances*. Each instance points to a *bottom-level acceleration structure* (BLAS) and includes an affine transformation matrix for the instanced BLAS, a 24-bit

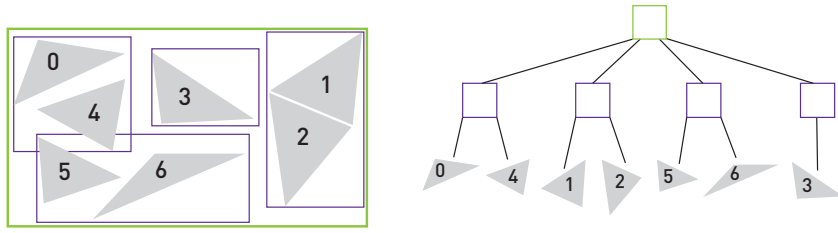


Figure 16-4. *Left: a possible bottom-level acceleration structure of seven triangle primitives using a bounding volume hierarchy. Right: a representation of the BLAS as a tree.*

instance ID, and a 32-bit shader offset index. Multiple instances can point to the same BLAS. A bottom-level acceleration structure can be an acceleration structure of triangles or an acceleration structure of axis-aligned bounding boxes (AABBs). The former type of BLAS is good for meshes. Each AABB in the latter type of BLAS represents a procedural object, defined by the corresponding intersection shader, instead of a mesh. Whenever a ray intersects an AABB, the device will call the corresponding intersection shader to determine if and where the ray intersects the procedural object (Figure 16-5).

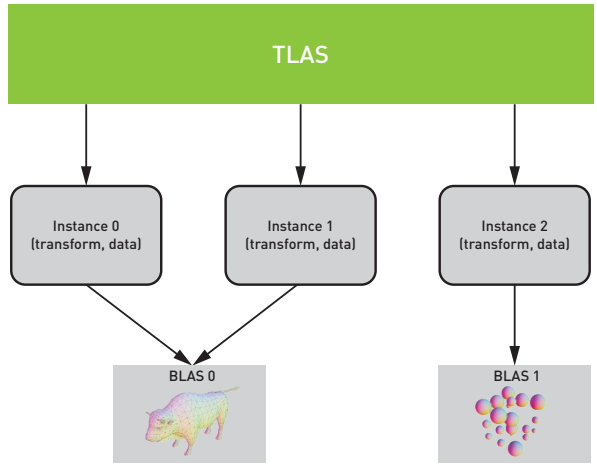


Figure 16-5. *A top-level acceleration structure of three instances pointing to either of two bottom-level structures. BLAS 0 contains triangle geometry, and BLAS 1 contains AABBs that bound custom sphere primitives (using an intersection shader). The TLAS contains two instances of BLAS 0 and one instance of BLAS 1.*

This two-level approach provides a good balance between acceleration structure build flexibility and traversal performance. For instance, individual BLASs of deforming meshes can be rebuilt without rebuilding the entire scene, and animating instance transformations only requires rebuilding the TLAS.

To build a BLAS or TLAS, describe the objects (such as triangles, AABBs, or instances), then call `vkCmdBuildAccelerationStructuresKHR()` to build the acceleration structure. This uses the GPU to generate the acceleration structure. Because there are many ways to specify and represent geometry, describing the scene is thus the primary initial challenge when building an acceleration structure.

Because geometry can change over time, acceleration structures often need to be modified. When an acceleration structure's objects change, applications have a choice between *rebuild* and *refit* operations. Rebuilding is required whenever the topology or number of primitives changes. Refitting is an option when moving primitives in an acceleration structure, but not adding or removing any. Refitting preserves the tree structure and is faster than rebuilding, but might result in slower ray tracing operations than rebuilding.

The next subsections show how to build acceleration structures and also provide an overview of refitting and compacting acceleration structures.

BOTTOM-LEVEL ACCELERATION STRUCTURE CONSTRUCTION

Both BLAS and TLAS construction involves six steps:

1. Get the host or device addresses of the geometry's buffers.
2. Describe the instance's geometry using one or more `VkAccelerationStructureGeometryKHR` objects (pointing to triangle, AABB, or instance primitives) and a `VkAccelerationStructureBuildRangeInfoKHR` object (giving the number of primitives and offsets for building).
3. Determine the worst-case memory requirements for the AS and for scratch storage required when building.
4. Create an empty acceleration structure and its underlying `VkBuffer`.
5. Allocate scratch space.

6. Call `vkCmdBuildAccelerationStructuresKHR()` with a populated `VkAccelerationStructureBuildSizesInfoKHR` struct and range info to build the geometry into an acceleration structure.

These functions can also be used to build multiple acceleration structures at once or to build acceleration structures on the host instead of the device. Host builds can be useful for further parallelizing acceleration structure builds across the system, but are not supported on all implementations.

The sample code in Listings 16-9–16-15 will show how to build a single BLAS and TLAS on a device, after uploading a mesh's vertex and index buffers to the device and obtaining their device addresses using `vkGetBufferDeviceAddress()`.

Start by creating a `VkAccelerationStructureGeometryTrianglesDataKHR()` structure (Listing 16-9) that specifies where the builder can find the vertices and optional indices for this triangle data, as well as their formats and lengths. Its `transformData` field can point to a `VkTransformMatrixKHR()` object (representing a 3×4 affine transformation matrix) on the host or device used to apply a transformation to the vertices before building the BLAS, or the `transformData` field can be zero-initialized to represent the identity transform.

Listing 16-9. *Specifying triangle mesh data using `VkAccelerationStructureGeometryTrianglesDataKHR`.*

```
std::vector<float>    objVertices   = ...;
std::vector<uint32_t> objIndices    = ...;
VkDeviceAddress vertexBufferAddress = ...;
VkDeviceAddress indexBufferAddress  = ...;

VkAccelerationStructureGeometryTrianglesDataKHR triangles{
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_TRIANGLES_DATA_KHR};
triangles.vertexFormat             = VK_FORMAT_R32G32B32_SFLOAT;
triangles.vertexData.deviceAddress = vertexBufferAddress;
triangles.vertexStride             = 3 * sizeof(float);
triangles.indexType                = VK_INDEX_TYPE_UINT32;
triangles.indexData.deviceAddress  = indexBufferAddress;
triangles.maxVertex                = uint32_t(objVertices.size() - 1);
triangles.transformData            = {0};  // No transform
```

Additionally, a triangle can be marked as *inactive* (as opposed to *active*) by setting the *X*-component of each vertex to a floating-point NaN. Similarly, an AABB can be marked as inactive by setting the *X*-component of its minimum vertex to a floating-point NaN. If an object is inactive, it will be considered invisible to all rays and should not be represented in the acceleration

structure. However, switching objects between being active and inactive after acceleration structure creation requires a rebuild instead of a refit (see Section 16.8.2).

Point to this VkAccelerationStructureGeometryTrianglesDataKHR struct using a VkAccelerationStructureGeometryKHR struct, which implements a polymorphic class in C, representing either triangle, AABB, or instance data. Also, add a flag to disable any-hit shaders on this geometry for faster ray tracing performance (Listing 16-10).

Listing 16-10. *Encapsulating geometry data in VkAccelerationStructureGeometryKHR.*

```
VkAccelerationStructureGeometryKHR geometry{
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR};
geometry.geometryType       = VK_GEOMETRY_TYPE_TRIANGLES_KHR;
geometry.geometry.triangles = triangles;
geometry.flags              = VK_GEOMETRY_OPAQUE_BIT_KHR;
```

Next, create a VkAccelerationStructureBuildRangeInfoKHR object that specifies the first vertex to build from, the number of primitives (triangles, AABBs, or instances) in the AS, and the offsets in bytes into the vertex, index, and transformation matrix buffers (Listing 16-11).

Listing 16-11. *Listing ranges of data to access in VkAccelerationStructureBuildRangeInfoKHR.*

```
VkAccelerationStructureBuildRangeInfoKHR rangeInfo;
rangeInfo.firstVertex     = 0;
rangeInfo.primitiveCount  = uint32_t(objIndices.size() / 3);
rangeInfo.primitiveOffset = 0;
rangeInfo.transformOffset = 0;
```

To query the worst-case amount of memory needed for the acceleration structure and for scratch space, partially specify a VkAccelerationStructureBuildGeometryInfoKHR struct, and then call vkGetAccelerationStructureBuildSizesKHR() (Listing 16-12). VkAccelerationStructureBuildGeometryInfoKHR points to an array of geometries to be built into the BLAS, as well as build settings. In this case, configure the settings to refer to a BLAS (instead of a TLAS), to build an acceleration structure (instead of refitting it), and to prefer fast ray tracing (over fast build times).

Next, create a buffer to use as the backing memory for the acceleration structure, using the ACCELERATION_STRUCTURE_STORAGE, SHADER_DEVICE_ADDRESS, and STORAGE_BUFFER usage bits. In this case,

Listing 16-12. *Partially specifying `VkAccelerationStructureBuildGeometryInfoKHR` and querying worst-case memory usage.*

```
VkAccelerationStructureBuildGeometryInfoKHR buildInfo{
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR};
buildInfo.flags = VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR;
buildInfo.geometryCount = 1;
buildInfo.pGeometries   = &geometry;
buildInfo.mode = VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR;
buildInfo.type = VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR;
buildInfo.srcAccelerationStructure = VK_NULL_HANDLE;
// We will set dstAccelerationStructure and scratchData once
// we have created those objects.

VkAccelerationStructureBuildSizesInfoKHR sizeInfo{
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_INFO_KHR};
vkGetAccelerationStructureBuildSizesKHR(
  // The device
  device,
  // Build on device instead of host.
  VK_ACCELERATION_STRUCTURE_BUILD_TYPE_DEVICE_KHR,
  // Pointer to build info
  &buildInfo,
  // Array of number of primitives per geometry
  &rangeInfo.primitiveCount,
  // Output pointer to store sizes
  &sizeInfo);
```

`allocator` is a memory allocator, an instance of the `nvvk::AllocatorDma` memory sub-allocator class from NVIDIA DesignWorks Samples [11]. Then, create an acceleration structure which utilizes the buffer using `vkCreateAccelerationStructureKHR()`, and add this to the information in `buildInfo` (Listing 16-13).

Listing 16-13. *Allocating an acceleration structure.*

```
// Allocate a buffer for the acceleration structure.
bufferBLAS = allocator.createBuffer(
  sizeInfo.accelerationStructureSize,
  VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR
  | VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT
  | VK_BUFFER_USAGE_STORAGE_BUFFER_BIT);

// Create an empty acceleration structure object.
VkAccelerationStructureCreateInfoKHR createInfo{
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR};
createInfo.type   = buildInfo.type;
createInfo.size   = sizeInfo.accelerationStructureSize;
createInfo.buffer = bufferBLAS.buffer;
createInfo.offset = 0;
NVVK_CHECK(vkCreateAccelerationStructureKHR(
  device, &createInfo, nullptr, &blas));

buildInfo.dstAccelerationStructure = blas;
```

Then, allocate the buffer for scratch data, and use its device address in `buildInfo` (Listing 16-14).

Listing 16-14. *Creating the scratch buffer.*

```
nvvk::Buffer scratchBuffer = allocator.createBuffer(
  sizeInfo.buildScratchSize,
    VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT
  | VK_BUFFER_USAGE_STORAGE_BUFFER_BIT);
buildInfo.scratchData.deviceAddress
  = GetBufferDeviceAddress(device, scratchBuffer.buffer);
```

Finally, call `vkCmdBuildAccelerationStructuresKHR()` with an `infoCount` of 1 to record a command to build the acceleration structure. As this function takes an array of pointers to `VkAccelerationStructureBuildRangeInfoKHR` objects, a pointer to `rangeInfo` is obtained beforehand (Listing 16-15).

Listing 16-15. *Calling `vkCmdBuildAccelerationStructuresKHR`.*

```
VkAccelerationStructureBuildRangeInfoKHR* pRangeInfo = &rangeInfo;
vkCmdBuildAccelerationStructuresKHR(
  cmdBuffer,     // The command buffer to record the command
  1,             // Number of acceleration structures to build
  &buildInfo,    // Array of ...BuildGeometryInfoKHR objects
  &pRangeInfo); // Array of ...RangeInfoKHR objects
```

When the command buffer completes, the acceleration structure will have been built.

Multiple acceleration structure builds can be submitted and run in parallel. However, two acceleration structure builds cannot use the same scratch space or destination acceleration structure at the same time. This means that two options for building multiple acceleration structures are to allocate the maximum scratch space needed for any individual acceleration structure and then build acceleration structures using pipeline barriers to serialize builds, or to allocate enough scratch space for all acceleration structures to build at once and remove the pipeline barriers.

Acceleration structure build information can also be specified by the device using `vkCmdBuildAccelerationStructuresIndirectKHR()` if it supports indirect builds.

TOP-LEVEL ACCELERATION STRUCTURE CONSTRUCTION

A TLAS is built from instances. Each instance contains an address to a BLAS, so we must first get the device address of one or more BLASs. In Listing 16-16, our TLAS only contains a single instance.

Listing 16-16. *Obtaining the device address of an acceleration structure.*

```
VkAccelerationStructureDeviceAddressInfoKHR addressInfo{
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_DEVICE_ADDRESS_INFO_KHR};
addressInfo.accelerationStructure = blas;
VkDeviceAddress blasAddress
  = vkGetAccelerationStructureDeviceAddressKHR(
      device, &addressInfo);
```

Listing 16-17 shows how to specify an instance using a `VkAccelerationStructureInstanceKHR` structure. The `transform` field of this structure is a 3×4 affine transformation matrix. It can be used to instance BLASs with different transforms; the effect is as if each (x, y, z) vertex was replaced by

$$M \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} M_{00}x + M_{01}y + M_{02}z + M_{03} \\ M_{10}x + M_{11}y + M_{12}z + M_{13} \\ M_{20}x + M_{21}y + M_{22}z + M_{23} \end{pmatrix}, \tag{16.1}$$

where M is the 3×4 matrix contained in `transform`. The `instanceCustomIndex` field stores an arbitrary 24-bit integer that shaders can access. The mask field stores an 8-bit integer; a ray can intersect an instance only if the bitwise `AND` of this mask and the ray's mask is nonzero. The field `instanceShaderBindingTableRecordOffset` contains an offset applied when looking up the instance's shaders in a shader binding table. The flags field enables certain features; here, it is used to disable backface culling. Finally, the `accelerationStructureReference` field contains the address of the instance's BLAS.

Listing 16-17. *Specifying an instance.*

```
// Zero-initialize.
VkAccelerationStructureInstanceKHR instance{};
// Set the instance transform to a 135-degree rotation around
// the y-axis.
const float rcpSqrt2 = sqrtf(0.5f);
instance.transform.matrix[0][0] = -rcpSqrt2;
instance.transform.matrix[0][2] = rcpSqrt2;
instance.transform.matrix[1][1] = 1.0f;
instance.transform.matrix[2][0] = -rcpSqrt2;
instance.transform.matrix[2][2] = -rcpSqrt2;
instance.instanceCustomIndex    = 0;
instance.mask                   = 0xFF;
instance.instanceShaderBindingTableRecordOffset = 0;
instance.flags = VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR;
instance.accelerationStructureReference = blasAddress;
```

Similar to how triangle data was stored on the device when using it to create a BLAS, create a buffer of instances on the device and upload the array of instances to it (Listing 16-18). This also means that the instance buffer could be written by the GPU, instead of the CPU. As before, use the `nvvk::AllocatorDma` memory allocator abstraction.

Listing 16-18. *Uploading an instance buffer of one instance to the `VkDevice` and waiting for it to complete.*

```
nvvk::Buffer bufferInstances = allocator.createBuffer(
  cmdBuffer, sizeof(instance), &instance,
  VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT);

// Add a memory barrier to ensure that createBuffer's upload command
// finishes before starting the TLAS build.
VkMemoryBarrier barrier{VK_STRUCTURE_TYPE_MEMORY_BARRIER};
barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
barrier.dstAccessMask = VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR;
vkCmdPipelineBarrier(cmdBuffer,
  VK_PIPELINE_STAGE_TRANSFER_BIT,
  VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
  0,
  1, &barrier,
  0, nullptr,
  0, nullptr);
```

Also, create a `VkAccelerationStructureBuildRangeInfoKHR` structure (described previously) as in Listing 16-19.

Listing 16-19. *Specifying range information for the TLAS build.*

```
VkAccelerationStructureBuildRangeInfoKHR rangeInfo;
rangeInfo.primitiveOffset = 0;
rangeInfo.primitiveCount  = 1; // Number of instances
rangeInfo.firstVertex     = 0;
rangeInfo.transformOffset = 0;
```

Next, much like how triangle data was pointed to with a `VkAccelerationStructureGeometryTrianglesDataKHR` structure encapsulated within a `VkAccelerationStructureGeometryKHR` structure when building a BLAS, point to the instance buffer using a `VkAccelerationStructureGeometryInstancesDataKHR` structure and also encapsulate it within a `VkAccelerationStructureGeometryKHR` structure. Here, `GetBufferDeviceAddress()` is a function that returns a buffer's device address using `vkGetBufferDeviceAddress()` (Listing 16-20).

Listing 16-20. *Constructing a `VkAccelerationStructureGeometryKHR` struct of instances.*

```
VkAccelerationStructureGeometryInstancesDataKHR instancesVk{
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_INSTANCES_DATA_KHR};
instancesVk.arrayOfPointers    = VK_FALSE;
instancesVk.data.deviceAddress = GetBufferDeviceAddress(device,
      bufferInstances.buffer);

// Like creating the BLAS, point to the geometry (in this case, the
// instances) in a polymorphic object.
VkAccelerationStructureGeometryKHR geometry
{VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR};
geometry.geometryType       = VK_GEOMETRY_TYPE_INSTANCES_KHR;
geometry.geometry.instances = instancesVk;
```

Finally, create the TLAS from the `VkAccelerationStructureGeometryKHR` and `VkAccelerationStructureBuildRangeInfoKHR` structures in the same way that the BLAS was built, with the exception of setting `VkAccelerationStructureBuildGeometryInfoKHR::type` to `VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR` (Listing 16-21).

Listing 16-21. *Allocating and building a top-level acceleration structure.*

```
// Create the build info: in this case, pointing to only one
// geometry object.
VkAccelerationStructureBuildGeometryInfoKHR buildInfo{
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR};
buildInfo.flags = VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR;
buildInfo.geometryCount = 1;
buildInfo.pGeometries   = &geometry;
buildInfo.mode = VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR;
buildInfo.type = VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR;
buildInfo.srcAccelerationStructure = VK_NULL_HANDLE;

// Query the worst-case AS size and scratch space size based on
// the number of instances (in this case, 1).
VkAccelerationStructureBuildSizesInfoKHR sizeInfo{
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_INFO_KHR};
vkGetAccelerationStructureBuildSizesKHR(
  device,
  VK_ACCELERATION_STRUCTURE_BUILD_TYPE_DEVICE_KHR,
  &buildInfo,
  &rangeInfo.primitiveCount,
  &sizeInfo);

// Allocate a buffer for the acceleration structure.
bufferTLAS = allocator.createBuffer(
  sizeInfo.accelerationStructureSize,
  VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR
  | VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT
  | VK_BUFFER_USAGE_STORAGE_BUFFER_BIT);

// Create the acceleration structure object.
// (Data has not yet been set.)
VkAccelerationStructureCreateInfoKHR createInfo  {
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR};
```

```
createInfo.type   = buildInfo.type;
createInfo.size   = sizeInfo.accelerationStructureSize;
createInfo.buffer = bufferTLAS.buffer;
createInfo.offset = 0;
NVVK_CHECK(vkCreateAccelerationStructureKHR(
  device, &createInfo, nullptr, &tlas));

buildInfo.dstAccelerationStructure = tlas;

// Allocate the scratch buffer holding temporary build data.
nvvk::Buffer bufferScratch = allocator.createBuffer(
  sizeInfo.buildScratchSize,
  VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT
  | VK_BUFFER_USAGE_STORAGE_BUFFER_BIT);
buildInfo.scratchData.deviceAddress
  = GetBufferDeviceAddress(device, bufferScratch.buffer);

// Create a one-element array of pointers to range info objects.
VkAccelerationStructureBuildRangeInfoKHR* pRangeInfo = &rangeInfo;

// Build the TLAS.
vkCmdBuildAccelerationStructuresKHR(
  cmdBuffer,
  1, &buildInfo,
  &pRangeInfo);
```

16.8.2 ACCELERATION STRUCTURE OPERATIONS

Vulkan supports several acceleration structure operations other than building an acceleration structure from geometry. The following subsections give brief overviews of cloning, refitting (updating), compacting, serializing, and deserializing acceleration structures (BLASs or TLASs):

- An acceleration structure can be *cloned* to another acceleration structure, copying its data.
- If an acceleration structure's geometry (triangles, AABBs, or instances) deforms but the amount of geometry stays the same, the acceleration structure can be *refit* to (or *updated* with) the new geometry. This is faster than rebuilding the acceleration structure, but the updated structure may not be as efficient to trace as a rebuilt structure. This is useful for animated meshes and scenes.
- An acceleration structure can be *compacted* by copying it with a compaction flag set to another acceleration structure. This allows the acceleration structure to take up less memory than allocated by the worst-case queried amount, unless the worst-case bound was tight.
- An acceleration structure can be *serialized* to a `VkBuffer`. This buffer can then be *deserialized* on a (possibly different) device to obtain an

equivalent acceleration structure. For instance, imagine a system where *N* devices path-trace an image in parallel: the acceleration structures used in the scene could be built in parallel, and then each acceleration structure could be serialized, transferred to, and then deserialized on the other *N* – 1 devices. Serialization can also be used for out-of-core ray tracing.

The next subsections will show how to perform these operations on the device; however, Vulkan also has functions that can be used to perform these operations on the host.

CLONING ACCELERATION STRUCTURES

To clone an acceleration structure, as in Listing 16-22, create a `VkCopyAccelerationStructureInfoKHR` structure, with `src` set to the acceleration structure to copy from, `dst` set to the acceleration structure to copy to, and `mode` set to `VK_COPY_ACCELERATION_STRUCTURE_MODE_CLONE_KHR`. Then, call `vkCmdCopyAccelerationStructureKHR()` with a command buffer to record to and a pointer to the `VkCopyAccelerationStructureInfoKHR` structure. The destination acceleration structure must have been created with the same parameters as the source acceleration structure.

Listing 16-22. *Cloning an acceleration structure.*

```
VkAccelerationStructure src, dst;
...
VkCopyAccelerationStructureInfoKHR copyInfo {
      VK_STRUCTURE_TYPE_COPY_ACCELERATION_STRUCTURE_INFO_KHR};
copyInfo.src  = src;
copyInfo.dst  = dst;
copyInfo.mode = VK_COPY_ACCELERATION_STRUCTURE_MODE_CLONE_KHR;

vkCmdCopyAccelerationStructureKHR(cmdBuf, &copyInfo);
```

REFITTING ACCELERATION STRUCTURES

If its geometry changes, an acceleration structure can be refit to the new geometry as long as the following are true:

- The acceleration structure was created with the `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_KHR` flag.
- The number of geometries has not changed.
- When updating a BLAS, only deformations have occurred.

- When updating a BLAS, the `geometry.triangles.transformData` member has not changed from `NULL` to non-`NULL`.
- All flag, format, range, and type members are the same as when the acceleration structure was last built [5].

For instance, the number of vertices and AABBs may not change, the contents of each call's index buffer (if any) must be identical, and no primitives may have switched between being active and being inactive (see page 229).

For an implementation that uses a BVH for an acceleration structure, for instance, this could correspond to refitting the nodes of the BVH to fit the new geometry, without modifying the BVH's tree structure or requiring a different amount of memory. This is usually faster than rebuilding the BVH, but the resulting structure might be less efficient for tracing than a rebuilt acceleration structure. In particular, refitting usually becomes less efficient compared to rebuilding as deformations increase in magnitude.

As a result, there are a number of strategies used in deciding when to rebuild or refit an acceleration structure. Sjoholm [13] recommends rebuilding TLASs, distributing BLAS rebuilds over frames, and rebuilding visible BLASs after large and unpredictable deformations, otherwise refitting or skipping updates entirely if visual errors can be avoided.

To update an acceleration structure, call `vkCmdBuildAccelerationStructuresKHR()` as was shown on pages 229 and 233, but with both `VkAccelerationStructureBuildGeometryInfoKHR`'s `srcAccelerationStructure` and `dstAccelerationStructure` fields set to the acceleration structure to update and with its mode field set to `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`.

COMPACTING ACCELERATION STRUCTURES

Building a compacted AS is a two-phase process. In the first phase, we build an AS into a worst-case-sized buffer, the same way we did before, with a flag allowing compaction to be enabled. In the second phase, we wait for the AS build to finish, query the compacted size, allocate an AS of the compacted size, copy the worst-case-size AS into the compacted AS, and then deallocate the worst-case-size AS.

In practice, it may be desirable to batch together multiple AS builds, in which case one may want to execute phase 1 on a set of acceleration structures

before moving to phase 2, rather than performing these phases sequentially for each individual AS.

We now describe the lower-level details (see also Listing 16-23). The first-phase acceleration structure must have been created with the VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_KHR flag set. Then, after recording the acceleration structure build command, record a pipeline barrier to wait for the build to complete, followed by a vkCmdWriteAccelerationStructuresPropertiesKHR() command with a pointer to a query pool of VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR queries to query the size of the acceleration structure if it were compacted. vkCmdWriteAccelerationStructuresPropertiesKHR() can also be used to query the compacted sizes of multiple acceleration structures at once and write the computed sizes into consecutive elements of the query pool.

Listing 16-23. *Computing the compacted size of one acceleration structure.*

```
// Create the query pool with space for one AS compacted size query.
VkQueryPoolCreateInfo qpci{VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO};
qpci.queryCount = 1;
qpci.queryType  = VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR;
VkQueryPool queryPool;
NVVK_CHECK(vkCreateQueryPool(
  device, &qpci, nullptr, &queryPool));

(build the acceleration structure...)

// Add a memory barrier to ensure that the acceleration structure build
// finishes before querying the compacted size.
VkMemoryBarrier barrier{VK_STRUCTURE_TYPE_MEMORY_BARRIER};
barrier.srcAccessMask = VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR;
barrier.dstAccessMask = VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR;
vkCmdPipelineBarrier(
  cmdBuf,
  VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
  VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
  0,
  1, &barrier,
  0, nullptr,
  0, nullptr);

// Write the compacted size to the query pool.
vkCmdWriteAccelerationStructuresPropertiesKHR(
  cmdBuf,     // Command buffer to record to
  1, &blas,   // Array of acceleration structures
  VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR,
  queryPool, // The query pool
  0);         // Index to start writing query results to

(end, submit, and wait for the command buffer...)
```

```
34
35 // Retrieve the size of the acceleration structure if compacted
36 VkDeviceSize compactedSize;
37 vkGetQueryPoolResults(
38   device,                    // The VkDevice
39   queryPool,                 // The query pool
40   0,                         // The index of the first query
41   1,                         // Number of queries
42   sizeof(VkDeviceSize),      // Size of buffer in bytes
43   &compactedSize,            // Pointer to buffer
44   sizeof(VkDeviceSize),      // Stride between elements
45   VK_QUERY_RESULT_WAIT_BIT); // Wait for queries to be available.
```

It is important to check that `compactedSize` is smaller than the original acceleration structure size. If this is the same as the original, worst-case size, then there's no need to create and copy the data to a compacted acceleration structure.

Next, create an acceleration structure with a buffer at least `compactedSize` bytes long. Finally, call `vkCmdCopyAccelerationStructureKHR()` (the same function used on page 238) with `VkCopyAccelerationStructureInfoKHR`'s `src` set to the uncompacted acceleration structure, `dst` set to the new acceleration structure, and mode set to `VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_KHR` (Listing 16-24).

Listing 16-24. *Creating a compacted copy of an acceleration structure.*

```
1 VkCopyAccelerationStructureInfoKHR copyInfo {
      VK_STRUCTURE_TYPE_COPY_ACCELERATION_STRUCTURE_INFO_KHR};
2 copyInfo.src = blas;
3 copyInfo.dst = compactedBLAS;
4 copyInfo.mode = VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_KHR;
5 vkCmdCopyAccelerationStructureKHR(cmdBuffer, &copyInfo);
```

SERIALIZING AND DESERIALIZING ACCELERATION STRUCTURES

To serialize an acceleration structure, first obtain the size of the serialized data. To do this, create a query pool with queries of type `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_SERIALIZATION_SIZE_KHR`; call `vkCmdWriteAccelerationStructuresPropertiesKHR()` with the acceleration structure, the query pool, and a `queryType` of the above query type; then, read the sizes from the query pool as on page 239.

After allocating memory with at least the required size, call `vkCmdCopyAccelerationStructureToMemoryKHR()` with a `VkCopyAccelerationStructureToMemoryInfoKHR` structure whose `src` is set to the acceleration structure, `dst.deviceAddress` is set to the device address

of the memory, and `mode` is set to `VK_COPY_ACCELERATION_STRUCTURE_MODE_SERIALIZE_KHR`. It is then possible to retrieve the serialized data from device memory and use it elsewhere.

To deserialize this acceleration structure on another device, first ensure that the serialized format is compatible with the device. To do this, call `vkGetDeviceAccelerationStructureCompatibilityKHR()` with the device, a pointer to the serialized data, and a `VkAccelerationStructureCompatibilityKHR` variable in which to store the result, as shown in Listing 16-25.

Listing 16-25. *Determining if a serialized acceleration structure is compatible with a device.*

```
VkAccelerationStructureVersionInfoKHR versionInfo {
      VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_VERSION_INFO_KHR};
versionInfo.pVersionData = serializedStructure;

VkAccelerationStructureCompatibilityKHR isCompatible;
vkGetDeviceAccelerationStructureCompatibilityKHR(
  device, &versionInfo, &isCompatible);
```

The serialized acceleration structure is compatible if the `VkAccelerationStructureCompatibilityKHR` variable is then equal to `VK_ACCELERATION_STRUCTURE_COMPATIBILITY_COMPATIBLE_KHR`.

To deserialize the serialized data, allocate space for and upload the data to the device, then allocate an acceleration structure of at least the size of the original structure. Then, call `vkCmdCopyMemoryToAccelerationStructureKHR()` with a `VkCopyMemoryToAccelerationStructureInfoKHR` structure whose `src.deviceAddress` points to the serialized data on the device, `dst` is set to the destination acceleration structure, and `mode` is set to `VK_COPY_ACCELERATION_STRUCTURE_MODE_DESERIALIZE_KHR`.

DESCRIPTOR SET LAYOUTS AND PIPELINE LAYOUTS

Recall that in Vulkan, shaders use descriptors to access resources. Descriptors are contained in descriptor sets, which are allocated from descriptor pools. Descriptor set layouts are like function signatures, in that they say what types of descriptors pipelines can access using descriptor bindings. Finally, pipelines use pipeline layouts, which include descriptor set layouts and push constant ranges.

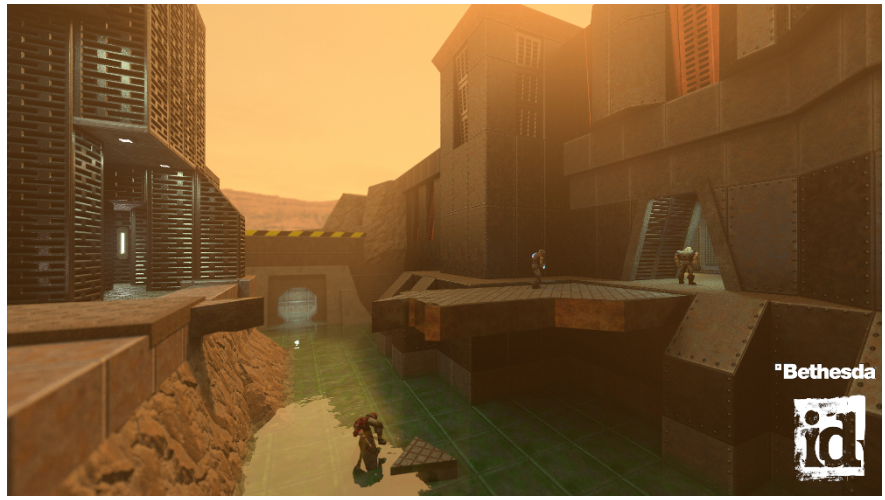


Figure 16-6. *Vulkan Ray Tracing on display in Quake II RTX.*

Vulkan Ray Tracing introduces several new shader stages. These are used when making descriptor bindings and push constants accessible by these shader stages, for instance. These new stage flags are the following:

- `VK_SHADER_STAGE_RAYGEN_BIT_KHR` (ray generation).
- `VK_SHADER_STAGE_ANY_HIT_BIT_KHR` (any-hit).
- `VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR` (closest-hit).
- `VK_SHADER_STAGE_MISS_BIT_KHR` (miss).
- `VK_SHADER_STAGE_INTERSECTION_BIT_KHR` (intersection).
- `VK_SHADER_STAGE_CALLABLE_BIT_KHR` (callable).

Additionally, Vulkan KHR Ray Tracing introduces a new descriptor type, `VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR`, for top-level acceleration structures.

The sample code in Listing 16-26 shows how to create a descriptor set layout with three storage buffers and one top-level acceleration structure, and how to create a descriptor pool with space for one descriptor set created using this layout. Generating a pipeline layout and allocating descriptor sets works the same as in Vulkan without ray tracing.

Listing 16-26. *Creating a Vulkan descriptor set layout and descriptor pool with ray tracing shaders and objects.*

```
// Descriptor set layout
// List all bindings:
// Position: 1 storage image, accessible from the raygen stage
// Normal: 1 storage image, accessible from the raygen stage
// AO: 1 storage image, accessible from the raygen stage
// TLAS: 1 acceleration structure, accessible from the raygen stage
std::array<VkDescriptorSetLayoutBinding, 4> bindings;

bindings[0].binding         = BINDING_IMAGE_POSITION;
bindings[0].descriptorType  = VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
bindings[0].descriptorCount = 1;
bindings[0].stageFlags      = VK_SHADER_STAGE_RAYGEN_BIT_KHR;

bindings[1]         = bindings[0];
bindings[1].binding = BINDING_IMAGE_NORMAL;

bindings[2]         = bindings[0];
bindings[2].binding = BINDING_IMAGE_AO;

bindings[3].binding         = BINDING_TLAS;
bindings[3].descriptorType  = VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
bindings[3].descriptorCount = 1;
bindings[3].stageFlags      = VK_SHADER_STAGE_RAYGEN_BIT_KHR;

VkDescriptorSetLayoutCreateInfo layoutInfo {
      VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO};
layoutInfo.bindingCount = uint32_t(bindings.size());
layoutInfo.pBindings    = bindings.data();
NVVK_CHECK(vkCreateDescriptorSetLayout(
  device, &layoutInfo, nullptr, &descriptorSetLayout));

// Descriptor pool with enough space for 1 set; this needs space for
// 3 storage image descriptors and 1 TLAS descriptor.
std::array<VkDescriptorPoolSize, 2> poolSizes;

poolSizes[0].type = VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
poolSizes[0].descriptorCount = 3;

poolSizes[1].type = VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
poolSizes[1].descriptorCount = 1;

VkDescriptorPoolCreateInfo descriptorPoolInfo {
      VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO};
descriptorPoolInfo.maxSets       = 1;
descriptorPoolInfo.poolSizeCount = uint32_t(poolSizes.size());
descriptorPoolInfo.pPoolSizes    = poolSizes.data();
NVVK_CHECK(vkCreateDescriptorPool(
  device, &descriptorPoolInfo, nullptr, &descriptorPool));
```

16.8.3 SHADER COMPILATION

In Vulkan, shaders are contained in SPIR-V shader modules, which can potentially contain entry points for more than one shader. SPIR-V is an

intermediate representation format, and SPIR-V shader modules can be generated in several ways, including from the following:

- GLSL code using a separate process such as `glslangValidator` or using a library such as `shaderc`.
- HLSL code using Microsoft's DirectX Shader Compiler (DXC).
- C++ code using Circle C++.
- Rust code using `rust-gpu`.

For instance, to translate a file named `ao.rgen` with a GLSL ray generation shader to SPIR-V bytecode contained in a file named `ao.rgen.spv`, call `glslangValidator` as follows:

```
glslangValidator --target-env vulkan1.2 -o ao.rgen.spv ao.rgen
```

Here, `glslangValidator` determines that the file represents a ray generation shader from the input file's `.rgen` extension. This can also be configured using `glslangValidator`'s `-S` flag.

A GLSL file may only have one entry point, which must be named `main`. However, entry points of SPIR-V modules can be renamed, modules can be linked together into single files, and other shading languages that can be translated to SPIR-V do not have this constraint.

Ray tracing shader stages can be specified using the same interface as other shader stages. A SPIR-V binary blob can be used to create a `VkShaderModule` as in Listing 16-27.

Listing 16-27. *Creating a Vulkan shader module from SPIR-V bytecode.*

```
1 const size_t    spirvSize = ...; // Length of SPIR-V bytecode in bytes
2 const uint32_t* spirvData = ...; // Pointer to SPIR-V data from file
3
4 VkShaderModuleCreateInfo createInfo {
      VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO};
5 createInfo.codeSize = spirvSize;
6 createInfo.pCode    = spirvData;
7
8 NVVK_CHECK(vkCreateShaderModule(
9   device, &createInfo, nullptr, &aoRgen));
```

16.9 CREATING VULKAN RAY TRACING PIPELINES

The shading stages of the Vulkan ray tracing pipeline were introduced earlier, but make up only part of the Vulkan ray tracing pipeline. Ray tracing pipelines

are also made up of fixed function traversal stages and a pipeline layout. This section will complement the earlier information and show how to configure the shaders, as well as the rest of the Vulkan rendering pipeline, in order to create a Vulkan ray tracing pipeline object. Ray tracing pipeline objects are a potentially large collection of ray generation, intersection, closest-hit, any-hit, and miss shaders coupled with a conglomerate of ray tracing–specific parameters. They are the ray tracing analogue of the graphics and compute pipeline objects, and they provide the runtime with the full set of shaders and configuration information for pipeline execution.

Vulkan pipeline creation takes several steps:

1. Load and compile shaders into `VkShaderModule` structures.
2. Aggregate shader entry points into an array of `VkPipelineStageCreateInfo` instances.
3. Create an array of `VkRayTracingShaderGroupCreateInfoKHR` structures. These will eventually be used to populate the shader binding table.
4. Use these two arrays as well as a pipeline layout to create a ray tracing pipeline object using `vkCreateRayTracingPipelinesKHR()`.

Sample code for loading and compiling shaders can be found in the previous section. Aggregation of the shader entry points from shader modules is shown in Listing 16-28.

Listing 16-28. *Creating a table of `VkPipelineShaderStageCreateInfo` objects. This uses three shader modules and points to the `RayGen()` entry point of the first shader module, the `Miss()` entry point of the second shader module, and the `CHS()` entry point of the third shader module.*

```
std::array<VkPipelineShaderStageCreateInfo, NumRtShaderStages> pssci{};

pssci[RayGenIndex].sType  =
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
pssci[RayGenIndex].module = primaryRayGenShader;
pssci[RayGenIndex].pName  = "RayGen";
pssci[RayGenIndex].stage  = VK_SHADER_STAGE_RAYGEN_BIT_KHR;

pssci[MissIndex].sType  =
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
pssci[MissIndex].module = missShader;
pssci[MissIndex].pName  = "Miss";
pssci[MissIndex].stage  = VK_SHADER_STAGE_MISS_BIT_KHR;

pssci[ClosestHitIndex].sType  =
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
pssci[ClosestHitIndex].module = closestHitShader;
pssci[ClosestHitIndex].pName  = "CHS";
pssci[ClosestHitIndex].stage  = VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR;
```

Listing 16-29. *Enumerating the elements of an array of shader groups, which contains one ray generation group, one miss group, and one hit group. The ray generation group lists the index of the ray generation shader stage. The miss shader group lists the index of the miss shader stage. The hit group lists the index of the closest-hit stage and does not use an any-hit or intersection shader.*

```
std::array<VkRayTracingShaderGroupCreateInfoKHR, NumRtShaderGroups> rtsgci
    {};

rtsgci[RayGenGroupIndex].sType =
    VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
rtsgci[RayGenGroupIndex].type  =
    VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
rtsgci[RayGenGroupIndex].generalShader      = PrimaryRayGenShaderIndex;
rtsgci[RayGenGroupIndex].closestHitShader   = VK_SHADER_UNUSED_KHR;
rtsgci[RayGenGroupIndex].anyHitShader       = VK_SHADER_UNUSED_KHR;
rtsgci[RayGenGroupIndex].intersectionShader = VK_SHADER_UNUSED_KHR;

// Miss groups also use the general group type.
rtsgci[MissGroupIndex] = rtsgci[RayGenGroupIndex];
rtsgci[MissGroupIndex].generalShader = MissShaderIndex;

// This hit group uses a TRIANGLES_HIT_GROUP group type.
rtsgci[HitGroupIndex].sType =
    VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR;
rtsgci[HitGroupIndex].type  =
    VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR;
rtsgci[HitGroupIndex].generalShader      = VK_SHADER_UNUSED_KHR;
rtsgci[HitGroupIndex].closestHitShader   = ClosestHitShaderIndex;
rtsgci[HitGroupIndex].anyHitShader       = VK_SHADER_UNUSED_KHR;
rtsgci[HitGroupIndex].intersectionShader = VK_SHADER_UNUSED_KHR;
```

This array describes the shader stages and will be used to populate a `VkRayTracingPipelineCreateInfoKHR` struct later in the code samples.

Next, populate the `VkRayTracingShaderGroupCreateInfoKHR` array (Listing 16-29). This describes the shader groups.

Finally, supply the two arrays as well as some additional parameters (pay close attention to `maxRecursionDepth` if you plan on performing recursive ray tracing) to the `VkRayTracingPipelineCreateInfoKHR` struct, and call `vkCreateRayTracingPipelinesKHR()` to create the ray tracing pipeline (Listing 16-30).

After creation, ray tracing pipeline objects can be bound and ray dispatch can commence, but not without the shader binding table, which will be discussed next.

Listing 16-30. *Creating the ray tracing pipeline.*

```
VkRayTracingPipelineCreateInfoKHR rtpci {
    VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_KHR};
rtpci.stageCount       = uint32_t(pssci.size());
rtpci.pStages          = pssci.data();
rtpci.groupCount       = uint32_t(rtsgci.size());
rtpci.pGroups          = rtsgci.data();
rtpci.maxRecursionDepth= 1;
rtpci.libraries.sType  = VK_STRUCTURE_TYPE_PIPELINE_LIBRARY_CREATE_INFO_KHR;
rtpci.layout           = pipelineLayout;

NVVK_CHECK(vkCreateRayTracingPipelinesKHR(
  device,          // The VkDevice
  VK_NULL_HANDLE,  // Don't request deferral
  1, &rtpci,       // Array of structures
  nullptr,         // Default host allocator
  &rtPipeline));   // Output VkPipelines
```



Figure 16-7. *Vulkan Ray Tracing in Wolfenstein: Youngblood.*

16.10 SHADER BINDING TABLES

The last core fundamental concept not yet discussed in detail is the shader binding table, commonly referred to by its acronym SBT. It is the mechanism that specifies which shaders will be executed when a ray intersects with a particular geometric instance and which resources are used given those conditions. This was introduced earlier in Section 16.2 as an array of shader references and associated interleaved per-object data that define the

relationship between the ray tracing shaders (or pipeline), their respective resources, and the scene geometry. In a typical scene, objects use a variety of shaders, materials, and resources. Because a ray has the ability to intersect with any object in the scene, all shaders and material data must be accessible from the SBT. This marks a significant departure from the raster graphics pipeline and workflow where draw submissions are commonly sorted by material.

This section is a brief introduction to SBTs in the context of Vulkan ray tracing. For a more comprehensive and thorough discussion of SBTs, see Chapter 15.

In Vulkan the SBT is a `VkBuffer` containing a group of records that consist of shader handles followed by in-line application-defined data. These records will be referred to here as *shader records*. Shader records in an SBT share a common stride, which must be large enough to accommodate the largest shader resource set. The shader handles determine which shaders are run for a given shader record, and the application-defined data is made available to the shaders as a shader storage buffer object (HLSL) or `shaderRecordEXT` buffer block (GLSL). Function call parameters, the shader pipeline, and acceleration structures together dictate which shader record will be used for a given circumstance. The application-defined data commonly includes resource indices, buffer device addresses, and numeric constants.

The example SBT in Figure 16-8 contains four shader records: one for the ray generation shader, one for the miss shader, and two hit groups. The hit groups correspond to different objects in the scene that have different shading information and are composed of a closest-hit shader and optional intersection and any-hit shaders. Though this SBT is usable, it is simpler than what is typically found in real ray tracing applications. SBTs are one of the most complicated parts of real-time ray tracing, and developers tend to run into problems here more frequently than in other places.

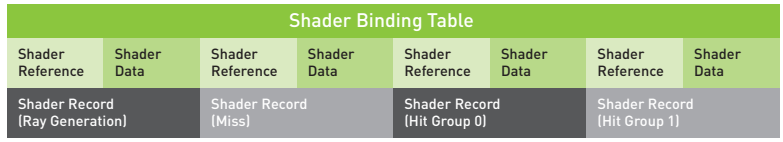


Figure 16-8. *Example shader binding table.*

The following is the formula used by the API for calculating the hit group record address in the shader binding table. It uses function parameters and acceleration structure data as inputs. Both bottom- and top-level acceleration structures can influence the calculation, and the function parameters can come from either the host side or from shaders:

$$\begin{aligned}\text{HitGroupRecordAddress} = \text{start} + \text{stride} * \big(&\text{instanceOffset} \\ &+ \text{paramOffset} + (\text{geometryIndex} * \text{paramMultiplier})\big)\,, \end{aligned} \quad (16.2)$$

where

- start is `((VkStridedBufferRegionKHR*)pHitShaderBindingTable)->offset`.
- stride is `((VkStridedBufferRegionKHR*)pHitShaderBindingTable)->stride`.
- instanceOffset is `VkAccelerationStructureInstanceKHR.instanceShaderBindingTableRecordOffset` (from the TLAS instance).
- paramOffset is `MultiplierForGeometryContributionToHitGroupIndex` (HLSL) or `sbtRecordOffset` (GLSL, from the parameter to `traceRayEXT()`).
- geometryIndex is the index of the geometry in the BLAS.
- paramMultiplier is `RayContributionToHitGroupIndex` (HLSL) or `sbtRecordStride` (GLSL, from the parameter to `traceRayEXT()`).

Listing 16-31 will help illustrate shader table creation. The first step is to compute the size of the SBT using the number of groups and the aligned handle size so that the buffer can be allocated.

Listing 16-31. *Computing a valid size and stride for the SBT.*

```
// The number of groups
auto groupCount = uint32_t(rayTracingShaderGroupCreateInfo.size());
// The size of a program identifier
uint32_t groupHandleSize = rtPipelineProperties.shaderGroupHandleSize;

// Compute the actual size needed per SBT entry by rounding up to the
// alignment needed. Nvh::align_up(a,b) rounds a up to a multiple of b.
uint32_t groupSizeAligned = nvh::align_up(groupHandleSize,
    rtPipelineProperties.shaderGroupBaseAlignment);

// Bytes needed for the SBT
uint32_t sbtSize = groupCount * groupSizeAligned;
```

The next step is to fetch the handles to the shader groups in the pipeline, allocate device memory for the SBT, and copy the handles into it (Listing 16-32).

Listing 16-32. *Allocating and writing shader handles from the ray tracing pipeline into the SBT.*

```
// Fetch all the shader handles used in the pipeline.
// This is opaque data, so we store it in a vector of bytes.
std::vector<uint8_t> shaderHandleStorage(sbtSize);
NVVK_CHECK(vkGetRayTracingShaderGroupHandlesKHR(
  device,               // The device
  rtPipeline,           // The ray tracing pipeline
  0,                    // Index of the group to start from
  groupCount,           // The number of groups
  sbtSize,              // Size of the output buffer in bytes
  shaderHandleStorage.data()); // The output buffer

// Allocate a buffer for storing the SBT.
rtSBTBuffer = allocator.createBuffer(
  sbtSize,
  VK_BUFFER_USAGE_TRANSFER_SRC_BIT
    | VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT
    | VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR,
  VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT
    | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);

// Map the SBT buffer and write in the handles.
void* mapped = allocator.map(rtSBTBuffer);
auto* pData = reinterpret_cast<uint8_t*>(mapped);
for(uint32_t g = 0; g < groupCount; g++)
{
  memcpy(pData,
    shaderHandleStorage.data() + g * groupHandleSize,
    groupHandleSize);
  pData += groupSizeAligned;
}
allocator.unmap(rtSBTBuffer);
```

There are a couple of items to be aware of regarding SBTs and their creation. First, there is no ordering requirement with respect to shader types in the SBT; ray generation, hit, and miss groups can come in any order. Next, it is important to pay close attention to alignment. There is no guarantee that group handle size will be the same as shader group alignment size.

In particular, here are the full requirements for SBT alignment, using fields from a `VkPhysicalDeviceRayTracingPipelinePropertiesKHR` object:

- The stride must be greater than `shaderGroupHandleSize` (in other words, each SBT record must have enough space for its shader group handle).
- The stride must be a multiple of `shaderGroupHandleAlignment`.

- The stride must be less than `maxShaderGroupStride`.

- The address of the first element in each `VkStridedDeviceAddressRegionKHR` region must be a multiple of `shaderGroupBaseAlignment` bytes.

16.11 RAY DISPATCH

Once the acceleration structures have been built, the shaders are loaded, the ray tracing pipeline is in place, and the shader binding table is in order, the application is finally ready to start dispatching rays. In order to do so, first bind the pipeline, as well as any descriptor sets and push constants that the pipeline requires (Listing 16-33).

Note that the listings in this section deviate from the sample the previous sections followed.

Listing 16-33. *Binding the ray tracing pipeline, descriptor set, and push constants.*

```
vkCmdBindPipeline(cmdBuffer,
  VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR,
  rtPipeline);

vkCmdBindDescriptorSets(cmdBuffer,
  VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR,
  rtPipelineLayout, 0, 1, &rtDescriptorSet, 0, 0);

vkCmdPushConstants(cmdBuffer, rtPipelineLayout,
  VK_SHADER_STAGE_ALL, 0,
  sizeof(rtConstants), &rtConstants);
```

The Vulkan command to dispatch rays, `vkCmdTraceRaysKHR()`, requires a command buffer handle, SBT regions for each shader stage type that it will use, and 3D grid dispatch dimensions. Each SBT region is specified by populating a `VkStridedBufferRegionKHR` struct and passing a pointer to that struct as a parameter to `vkCmdTraceRaysKHR()` (Listing 16-34).

The size of the dispatch can also be specified by the device using `vkCmdTraceRaysIndirectKHR()` if it supports indirect ray tracing pipeline dispatches.

Remember to include any necessary synchronization, such as pipeline barriers. An example pipeline barrier appears in Listing 16-35 for reference.

Listing 16-34. *Strided buffer region specification and ray dispatch.*

```
VkStridedBufferRegionKHR rayGenRegion = {};
rayGenRegion.buffer = shaderBindingTable;
rayGenRegion.offset = shaderTableRecordSize * RayGenGroupIndex;
rayGenRegion.size   = rayTracingProperties.shaderGroupHandleSize;

VkStridedBufferRegionKHR missRegion = {};
missRegion.buffer = shaderBindingTable;
missRegion.offset = shaderTableRecordSize * MissGroupIndex;
missRegion.size   = rayTracingProperties.shaderGroupHandleSize;

VkStridedBufferRegionKHR hitRegion = {};
hitRegion.buffer = shaderBindingTable;
hitRegion.offset = shaderTableRecordSize * HitGroupIndex;
hitRegion.size   = shaderTableRecordSize * numHitRecords;
hitRegion.stride = shaderTableRecordSize;

VkStridedBufferRegionKHR callableRegion = {};

vkCmdTraceRaysKHR(cmdBuffer,
  &rayGenRegion,
  &missRegion,
  &hitRegion,
  &callableRegion,
  width, height, 1);
```

Listing 16-35. *Recording a memory barrier between shader writes from the ray tracing pipeline stage and shader reads from the compute shader pipeline stage.*

```
// Make compute shader stages wait for ray tracing to complete.
VkMemoryBarrier memoryBarrier{VK_STRUCTURE_TYPE_MEMORY_BARRIER};
memoryBarrier.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
memoryBarrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

vkCmdPipelineBarrier(cmdBuffer,
  VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR,
  VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
  0,
  1, &memoryBarrier,
  0, nullptr,
  0, nullptr);
```

16.12 ADDITIONAL RESOURCES

Even though Vulkan Ray Tracing is relatively new, there are many resources to supplement this introduction. There are samples available at the Khronos samples repository [4] that are very accessible to beginners. The NVIDIA DesignWorks samples collection [11] has various samples with different levels of complexity [2, 6, 7], which may help mitigate the steep learning curve encountered while moving beyond the smallest examples. There are also developer blogs online that discuss topics such as best practices, avoiding

potential development pitfalls, and leveraging tools to make development and debugging easier [10, 13].

Examination of an existing implementation such as Falcor [1], or even a larger game engine, may prove to be insightful and may help chart the course for implementing a new framework. There is ample opportunity to make large architectural contributions in this regard.

One potentially great way to extend Vulkan Ray Tracing knowledge would be to examine a basic tutorial and to augment it with another effect, such as by adding shadows or reflections to a simple example. Taking bite-size steps when beginning will prevent becoming overwhelmed by the complexity inherent in this subject matter. Lastly, contributing to community efforts, adding to the Vulkan specification, filing bugs, or any other way of getting involved and immersed in the ecosystem is a great way to learn and to help others to learn as well.

16.13 CONCLUSION

Vulkan coupled with the ray tracing extensions provides an enormous amount of capability for the ambitious programmer. This chapter served as a basic introduction to the fundamentals needed to get a Vulkan Ray Tracing application up and running. Even reduced to the bare minimum, Vulkan Ray Tracing applications contain significant complexity. After becoming familiar with the basics, it is highly recommended that developers keep up to date with the latest developments to the Vulkan API as well as tools that facilitate debugging and analysis of applications.

ACKNOWLEDGMENTS

Thank you to Martin-Karl Lefrançois for providing assistance with the sample and for building the original version of the acceleration structure creation code in Section 16.8.1. Mathias Heyer and Detlef Roettger provided clarification regarding the possibility of parallel BLAS builds.

REFERENCES

[1] Benty, N., Yao, K.-H., Clarberg, P., Chen, L., Kallweit, S., Foley, T., Oakes, M., Lavelle, C., and Wyman, C. The Falcor real-time rendering framework. https://github.com/NVIDIAGameWorks/Falcor, Aug. 2020.

[2] Bickford, N. vk_mini_path_tracer. https://github.com/nvpro-samples/vk_mini_path_tracer, Dec. 14, 2020. Accessed May 26, 2021.

[3] Khronos Group. GLSL_EXT_ray_tracing. https://github.com/KhronosGroup/GLSL/blob/master/extensions/ext/GLSL_EXT_ray_tracing.txt, Nov. 20, 2020. Accessed Feb. 1, 2021.

[4] Khronos Group. Vulkan-samples. https://github.com/KhronosGroup/Vulkan-Samples. Accessed Feb. 1, 2021.

[5] Khronos Group. Vulkan specification. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/vkspec.html. Accessed Feb. 1, 2021.

[6] Lefrançois, M.-K. OpenGL Interop—Raytracing. https://github.com/nvpro-samples/gl_vk_raytrace_interop, Dec. 3, 2019. Accessed Feb. 1, 2021.

[7] Lefrançois, M.-K., Gautron, P., Bickford, N., and Akeley, D. NVIDIA Vulkan Ray Tracing tutorials. https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/, Mar. 31, 2020. Accessed Feb. 1, 2021.

[8] LunarG. Vulkan. https://vulkan.lunarg.com/. Accessed Feb. 23, 2021.

[9] Microsoft. DirectX shader compiler. https://github.com/Microsoft/DirectXShaderCompiler. Accessed Feb. 1, 2021.

[10] Mihut, A. Exploring ray tracing techniques in Wolfenstein: Youngblood. *Khronos Group*, https://www.khronos.org/blog/vulkan-ray-tracing-best-practices-for-hybrid-rendering, Nov. 23, 2020. Accessed May 26, 2021.

[11] NVIDIA. NVIDIA DesignWorks samples. https://github.com/nvpro-samples. Accessed Feb. 23, 2021.

[12] NVIDIA. Nvpro-core. *NVIDIA DesignWorks Samples*, https://github.com/nvpro-samples/nvpro_core. Accessed Feb. 23, 2021.

[13] Sjoholm, J. Best practices: Using NVIDIA RTX ray tracing. *NVIDIA Developer Blog*, https://developer.nvidia.com/blog/best-practices-using-nvidia-rtx-ray-tracing/, Aug. 10, 2020. Accessed Feb. 23, 2021.

[14] Wächter, C. and Binder, N. A fast and robust method for avoiding self-intersection. In E. Haines and T. Akenine-Möller, editors, chapter 6, pages 77–85. Apress, 2019.

[15] Wyman, C. and Marrs, A. Introduction to DirectX Raytracing. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 3, pages 21–47. Apress, 2019.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 17

USING BINDLESS RESOURCES WITH DIRECTX RAYTRACING

Matt Pettineo
Ready At Dawn Studios

ABSTRACT

Resource binding in Direct3D 12 can be complex and difficult to implement correctly, particularly when used in conjunction with DirectX Raytracing. This chapter will explain how to use bindless techniques to provide shaders with global access to all resources, which can simplify application and shader code while also enabling new techniques.

17.1 INTRODUCTION

Prior to the introduction of Direct3D 12, GPU texture and buffer resources were accessed using a simple CPU-driven binding model. The GPU's resource access capabilities were typically exposed as a fixed set of "slots" that were



Figure 17-1. *The Unreal Engine Sun Temple scene [5] rendered with an open source DXR path tracer [9] that uses bindless resources.*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_17

tied to a particular stage of the logical GPU pipeline, and API functions were provided that allowed the GPU to "bind" a resource view to one of the exposed slots. This sort of binding model was a natural fit for earlier GPUs, which typically featured a fixed set of hardware descriptor registers that were used by the shader cores to access resources.

While this old style of binding was relatively simple and well understood, it naturally came with many limitations. The limited nature of the binding slots meant that programs could typically only bind the exact set of resources that would be accessed by a particular shader program, which would often have to be done before every draw or dispatch. The CPU-driven nature of binding demanded that a shader's required resources had to be statically known after compilation, which naturally led to inherent restrictions on the complexity of a shader program.

As ray tracing on the GPU started to gain traction, the classic binding model reached its breaking point. Ray tracing tends to be an inherently global process: one shader program might launch rays that could potentially interact with every material in the scene. This is largely incompatible with the notion of having the CPU bind a fixed set of resources prior to dispatch. Techniques such as atlasing or Sparse Virtual Texturing [1] can be viable as a means of emulating global resource access, but may also require adding significant complexity to a renderer.

Fortunately, newer GPUs and APIs no longer suffer from the same limitations. Most recent GPU architectures have shifted to a model where resource descriptors can be loaded from memory instead of from registers, and in some cases they can also access resources directly from a memory address. This removes the prior restrictions on the number of resources that can be accessed by a particular shader program, and also opens the door for those shader programs to dynamically choose which resource is actually accessed.

This newfound flexibility is directly reflected in the binding model of Direct3D 12, which has been completely revamped compared to previous versions of the API. In particular, it supports features that collectively enable a technique commonly known as *bindless resources* [2]. When implemented, bindless techniques effectively provide shader programs with full global access to the full set of textures and buffers that are present on the GPU. Instead of requiring the CPU to bind a view for each individual resource, shaders can instead access an individual resource using a simple 32-bit index that can be

freely embedded in user-defined data structures. While this level of flexibility can be incredibly useful in more traditional rasterization scenarios [6], they are borderline essential when using DirectX Raytracing (DXR).

The remainder of this chapter will cover the details of how to enable bindless resource access using Direct3D 12 (D3D12), and will also cover the basics of how to use bindless techniques in a DXR ray tracer. Basic familiarity with both D3D12 and DXR is assumed, and we refer the reader to an introductory chapter from the first volume of *Ray Tracing Gems* [11].

17.2 TRADITIONAL BINDING WITH DXR

Like the rest of D3D12, DXR utilizes *root signatures* to specify how resources should be made available to shader programs. These root signatures specify collections of root descriptors, descriptor tables, and 32-bit constants and map those to ranges of HLSL binding registers. When using DXR, we actually deal with two different types of root signatures: a global root signature and a local root signature. The global root signature is applicable to the ray generation shader as well as all executed miss, any-hit, closest-hit, and intersection shaders. The local root signature only applies to a particular hit group. Used together, global and local root signatures can implement a fairly traditional binding model where the CPU code "pushes" descriptors for all resources that are needed by the shader program. In a typical rendering scenario, this would likely involve having the local root signature provide a descriptor table containing all textures and constant buffers required by the particular material assigned to the mesh in the hit group. An example of this traditional model of resource binding is shown in Figure 17-2.

Though this approach can be workable, there are several problems that make it less than ideal. First, the mechanics of the local root signature are somewhat inconsistent with how root signatures normally work within D3D12. Standard root signatures require using command list APIs to specify which constants, root descriptors, and descriptor table should be bound to corresponding entries in the root signature. Local root signatures do not work this way because there can be many different root signatures contained within a single state object. Instead, the parameters for the root signature entries must be placed inline within a shader record in the hit group shader table, immediately following the shader identifier. This setup is further complicated by the fact that both shader records and root signature parameters have specific alignment requirements that must be observed. Shader identifiers

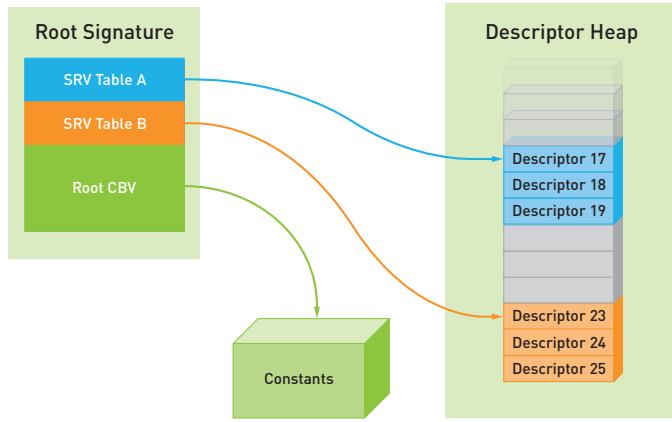


Figure 17-2. *An example of traditional resource binding in D3D12. A root signature contains two entries for Shader Resource View (SRV) descriptor tables, each of which point to a range of contiguous SRV descriptors within a global descriptor heap, as well as contains the root Constant Buffer View (CBV).*

consume 32 bytes (`D3D12_SHADER_IDENTIFIER_SIZE_IN_BYTES`) and must also be located at an offset that is aligned to 32 bytes (`D3D12_RAYTRACING_SHADER_RECORD_BYTE_ALIGNMENT`). Root signature parameters that are 8 bytes in size (such as root descriptors) must also be placed at offsets that are aligned to 8 bytes. Thus, carefully written packing code or helper types have to be used in order to fulfill these specific rules when generating the shader table. The following code shows an example of what shader record helper structs might look like:

```
struct ShaderIdentifier
{
    uint8_t Data[D3D12_SHADER_IDENTIFIER_SIZE_IN_BYTES] = { };

    ShaderIdentifier() = default;
    explicit ShaderIdentifier(const void* idPointer)
    {
        memcpy(Data, idPointer, D3D12_SHADER_IDENTIFIER_SIZE_IN_BYTES);
    }
};

struct HitGroupRecord
{
    ShaderIdentifier ID;
    D3D12_GPU_DESCRIPTOR_HANDLE SRVTableA = { };
    D3D12_GPU_DESCRIPTOR_HANDLE SRVTableB = { };
    uint32_t Padding1 = 0; // Ensure that CBV has 8-byte alignment.
    uint64_t CBV = 0;
    uint8_t Padding[8] = { }; // Needed to keep shader ID at
                              // 32-byte alignment
};
```

For more complex scenarios with many meshes and materials, the local root signature approach can quickly become unwieldy and error-prone. Generating the local root signature arguments on the CPU as part of filling the hit shader table is the most straightforward approach, but this can consume precious CPU cycles if it needs to be done frequently in order to support dynamic arguments. It also subjects us to some of the same limitations on our shader programs that we had in previous versions of D3D: the set of resources required for a hit shader must be known in advance, and the shader itself must be written so that it uses a static set of resource bindings. Generation of the shader table on the GPU using compute shaders is a viable option that allows for more GPU-driven rendering approaches, however this can be considerably more difficult to write, validate, and debug compared with the equivalent CPU implementation. It also does not remove the limitations regarding shader programs and dynamically selecting which resources to access. Both approaches are fundamentally incompatible with the new *inline raytracing* functionality that was added for DXR 1.1, since using a local root signature is no longer an option. This means that we must consider a less restrictive binding solution in order to make use of the new `RayQuery` APIs for general rendering scenarios.

17.3 BINDLESS RESOURCES IN D3D12

As mentioned earlier, bindless techniques allow us to effectively provide our shader programs with global access to all currently loaded resources instead of being restricted to a small subset. D3D12 supports bindless access to every resource type that utilizes shader visible descriptor heaps: Shader Resource Views (SRVs), Constant Buffer Views (CBVs), Unordered Access Views (UAVs), and Samplers. However, these types each have different limitations that can prevent their use in bindless scenarios, most of which are dictated by the value of `D3D12_RESOURCE_BINDING_TIER` that is exposed by the device. We will cover some of these limitations in more detail in Section 17.5, but for now we will primarily focus on using bindless techniques for SRVs because they typically form the bulk of resources accessed by shader programs. However, the concepts described here can generally be extended to other resource views with little effort.

They key to enabling bindless resources with D3D12 is setting up our root signature in a way that effectively exposes an entire descriptor heap through a single root parameter. The most straightforward way to do this is to add a parameter with a type of `D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE`,

with a single unbounded descriptor range:

```
// Unbounded range of descriptor SRV to expose the entire heap
D3D12_DESCRIPTOR_RANGE1 srvRanges[1] = {};
srvRanges[0].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_SRV;
srvRanges[0].NumDescriptors = UINT_MAX;
srvRanges[0].BaseShaderRegister = 0;
srvRanges[0].RegisterSpace = 0;
srvRanges[0].OffsetInDescriptorsFromTableStart = 0;
srvRanges[0].Flags = D3D12_DESCRIPTOR_RANGE_FLAG_DESCRIPTORS_VOLATILE;

D3D12_ROOT_PARAMETER1 params[1] = {};

// Descriptor table root parameter
params[0].ParameterType = D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;
params[0].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;
params[0].DescriptorTable.pDescriptorRanges = srvRanges;
params[0].DescriptorTable.NumDescriptorRanges = 1;
```

When building our command lists for rendering, we can then pass the handle returned by `ID3D12DescriptorHeap::GetGPUDescriptorHandleForHeapStart()` to `ID3D12GraphicsCommandList::SetGraphicsRootDescriptorTable()` or `ID3D12GraphicsCommandList::SetComputeRootDescriptorTable()` in order to make the entire contents of that heap available to the shader. This allows us to place our created descriptors anywhere in the heap without needing to partition it in any way.

To access a particular resource's descriptor in our shader program, we can use a technique known as *descriptor indexing*. In HLSL, this technique first requires us to declare an array of a particular shader resource type (such as `Texture2D`). To access a particular resource, we merely need to index into the array using an ordinary integer. A simple and straightforward way to do this is to use a constant buffer to pass descriptor indices to a shader, as demonstrated in Figure 17-3.

```
Texture2D GlobalTextureArray[] : register(t0);
SamplerState MySampler : register(s0);

struct MyConstants
{
    uint TexDescriptorIndex;
};
ConstantBuffer<MyConstants> MyConstantBuffer : register(b0);

float4 MyPixelShader(in float2 uv : UV) : SV_Target0
{
    uint texDescriptorIndex = MyConstantBuffer.TexDescriptorIndex;
    Texture2D myTexture = GlobalTextureArray[texDescriptorIndex];
    return myTexture.Sample(MySampler, uv);
}
```

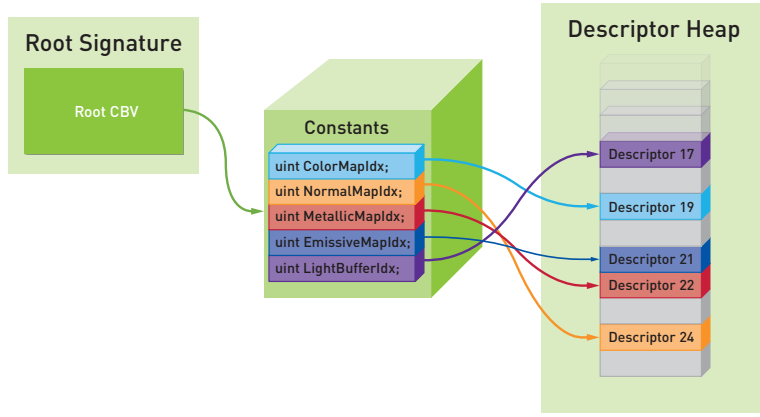


Figure 17-3. *An example of using bindless techniques to access SRV descriptors. A root signature contains a root Constant Buffer View, which points to a block of constants containing 32-bit indices of descriptors within a global descriptor heap. With a bindless setup the descriptors needed by a shader do not need to be contiguous within the descriptor heap and do not need to be ordered with regards to how the shader accesses or declares the resources.*

For this example to work, we simply need to ensure that our root signature's descriptor table parameter is mapped to the `t0` register used by the `Texture2D` array in our shader. If this is done properly, the shader effectively has full access to the entire global descriptor heap. Or, at least it can access all of the descriptors that can be mapped to the `Texture2D` HLSL type. One limitation of the current version of HLSL is that we need to declare a separate array for each HLSL resource type that we would like to access in our shader, and each one must have a separate non-overlapping register mapping. A simple way to ensure that their assignments don't overlap is to use a different register space for each resource array. This allows us to continue using unbounded arrays instead of requiring an array size to be compiled into the shader.

```
1 Texture2D Tex2DTable[]              : register(t0, space0);
2 Texture2D<uint4> Tex2DUintTable[]   : register(t0, space1);
3 Texture2DArray Tex2DArrayTable[]    : register(t0, space2);
4 TextureCube TexCubeTable[]          : register(t0, space3);
5 Texture3D Tex3DTable[]              : register(t0, space4);
6 Texture2DMS<float4> Tex2DMSTable[]  : register(t0, space5);
7 ByteAddressBuffer RawBufferTable[]  : register(t0, space6);
8 Buffer<uint> BufferUintTable[]      : register(t0, space7);
9 // ... and so on
```

Note that we not only need separate arrays for different resource types like `Texture2D` versus `Texture3D`, but we may also require having separate arrays

for different *return types* (expressed using the C++ template syntax) of the same HLSL resource type. This is evident in the previous example, which has arrays of both `Texture2D` as well as `Texture2D<uint4>` (a texture resource with no return type has an implicit return type of `float4`). Having textures with various return types is an unfortunate necessity for supporting all possible DirectX Graphics Infrastructure (DXGI) texture formats because certain formats require the shader to declare the HLSL resource with a specific return type. The following table lists the appropriate return type for each of the format modifiers available in the `DXGI_FORMAT` enumeration:

UNORM	float
SNORM	float
FLOAT	float
UINT	uint
SINT	int

One consequence of having separate HLSL resource arrays and register space bindings is that we must have a corresponding descriptor range in our root signature for each declared array. Luckily for us, it is possible to stack multiple unbounded descriptor ranges in a single root parameter. This ultimately means that we only need to bind our global descriptor heap once for each root signature:

```
D3D12_DESCRIPTOR_RANGE1 ranges[NumDescriptorRanges] = {};
for(uint32_t i = 0; i < NumDescriptorRanges; ++i)
{
    ranges[i].RangeType = D3D12_DESCRIPTOR_RANGE_TYPE_SRV;
    ranges[i].NumDescriptors = UINT_MAX;
    ranges[i].BaseShaderRegister = 0;
    ranges[i].RegisterSpace = i;
    ranges[i].OffsetInDescriptorsFromTableStart = 0;
    ranges[i].Flags = D3D12_DESCRIPTOR_RANGE_FLAG_DESCRIPTORS_VOLATILE;
}

D3D12_ROOT_PARAMETER1 params[1] = {};

// Descriptor table root parameter
params[0].ParameterType = D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE;
params[0].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;
params[0].DescriptorTable.pDescriptorRanges = ranges;
params[0].DescriptorTable.NumDescriptorRanges = 1;
```

With this approach, we can expand our support for bindless access to many types of buffer and texture resources. We can even declare our set of HLSL resource arrays in a single header file and then include that file in any shader code that needs to access resources. But what about types such as `StructuredBuffer` or `ConstantBuffer`, which are typically templated on a

user-defined struct type? There are effectively unlimited permutations of these types, which precludes us from predefining them in a shared header file. One possible approach for handling these types is to reserve some additional descriptor table ranges in the root signature that can be utilized by any individual shader program. As an example, we can define our root signature with eight additional descriptor ranges using register spaces 100 through 107. If we then write a shader program that needs to access a `StructuredBuffer` with a custom structure type, we simply declare an array bound to one of these reserved register spaces:

```
// Define an array of our custom buffer type assigned to
// one of our reserved register spaces.
StructuredBuffer<MyStruct> MyBufferArray[] : register(t0, space100);

MyStruct AccessMyBuffer(uint descriptorIndex, uint bufferIndex)
{
    StructuredBuffer<MyStruct> myBuffer = MyBufferArray[descriptorIndex];
    return myBuffer[bufferIndex];
}
```

In our original descriptor indexing example, we pulled our descriptor index from a constant buffer. This is a perfectly straightforward way for our CPU code to pass these indices to a shader program, and by filling out the constant buffer just before a draw or dispatch, we can even use this approach to emulate a more traditional binding setup. However, we are by no means limited to only using constant buffers for passing around descriptor indices. Because they are just a simple integer, we can now pack these almost anywhere. For instance, we could have a `StructuredBuffer` containing a set of descriptor indices for every loaded material in the scene, and a shader program could use a material index to fetch the appropriate set of indices. The indices could even be written into a UINT-formatted (unsigned integer) render target texture if that were useful! We can actually start to think of these indices as handles to our resources, or even as a pointer that grants us access to a resource's contents.

When writing shader code that uses descriptor indexing, we must always be careful to evaluate whether or not a particular index is *uniform*. In this context, a descriptor is uniform if it has the same value for all threads within a particular draw, dispatch, or hit, miss, any-hit, or intersection shader invocation.[1] In our original descriptor indexing example, the index came from a constant buffer, which means that all of the pixel shader threads used the

[1] Note that this definition is distinct from its meaning when used within the context of wave-level shader programming, where *uniform* means that the value is the same within a particular warp or wavefront.

same descriptor index. We would consider the index to be uniform in this case. However, let us now consider a more complex scenario. What if instead of coming from a constant buffer, the index was passed from the vertex shader as an interpolant? This would allow the value of that index to vary within the pixel shader threads. In this case the index would be considered *nonuniform*. Nonuniform descriptor indices are allowed in D3D12, however they do require special consideration. In order to ensure correct results, the index must be passed to the `NonUniformResourceIndex` intrinsic before being used to index into the resource array. This notifies the driver that the index may be varying, which may require it to insert additional instructions in order to properly handle the varying descriptor within a SIMD execution environment. On most existing GPU architectures, the additional cost of these instructions is proportional to the amount of divergence within the architecture-specific thread grouping (often referred to as a *warp* or *wavefront*). For these reasons it's important to be judicious about when and where to make use of nonuniform indexing. It's also important to be aware that the results of nonuniform indexing are undefined when `NonUniformResourceIndex` is omitted, which means that forgetting to use it may result in correct results on one architecture while producing graphical artifacts on others.

Though the additional flexibility afforded to our shaders can be a primary motivator for adopting bindless-style descriptor indexing, these techniques can also allow for simplification of application code when deployed throughout a rendering engine. Accessing a descriptor by indexing inherently grants us sparse access to all descriptors in a heap, which frees us from having to keep a particular shader's set of descriptors contiguous within that heap. Maintaining contiguous descriptor tables can often require complex (and expensive) management to be performed by our CPU code, which is especially true in cases where descriptors need to be updated in response to changing data.

A very common example is a CPU-updated buffer, often referred to as a *dynamic buffer* in earlier versions of D3D. Because the CPU cannot write to a buffer while the GPU is reading from it, techniques such as double-buffering or ring-buffering are often deployed to ensure that there is no concurrent access to the same resource memory. However, the act of "swapping" to a new internal buffer also requires swapping descriptors (if not using root descriptors), which causes any existing descriptor tables to become invalid. Dealing with this might normally require versioning entire descriptor tables,

or spending CPU cycles updating every table in which a particular buffer is referenced. With bindless techniques we no longer need to worry about contiguous descriptor tables, which means that a particular buffer only needs one descriptor to be updated within a heap whenever the buffer's contents change. We still need to be careful not to update a descriptor heap that's currently being referenced by executing GPU commands, however we can handle this at a global level by swapping through `N` global descriptor heaps (where `N` is the maximum number of frames in flight allowed by the renderer). As long as the descriptors for a buffer's "versions" are always placed at the same offset within the global heap, the shaders can continue to access the buffer using the same persistent descriptor index. For an example of how these patterns can be used in practice, consult the implementation of `StructuredBuffer::Map()` in the DXRPathTracer repository on GitHub [9].

17.4 BINDLESS RESOURCES WITH DXR

Though bindless techniques are very useful in general, their benefits really begin to become apparent when used in conjunction with ray tracing. Having global access to all texture and buffer resources is a great fit when tracing rays that can potentially intersect with any geometry in the entire scene, and is a de facto requirement when working with the new inline tracing functionality introduced with DXR 1.1. In this section, we will walk through an example implementation of a simple path tracer that utilizes bindless techniques to access geometry buffers as well as per-material textures. The source code and Visual Studio project for the complete implementation is available to view and download on GitHub [7].

Our simple path tracer will work as follows:

- For every frame, `DispatchRays` is called to launch one thread per pixel on the screen.
- Every dispatched thread traces a single camera ray into the scene.
- In the closest-hit shader, the surface properties are determined from geometry buffers and material textures.
- The hit shader computes direct lighting from the sun and local light sources, casting a shadow ray to determine visibility.
- The hit shader recursively traces another ray into the scene to gather indirect lighting from other surfaces.

- A miss shader is used to sample sky lighting from a procedural sky model.
- For materials using alpha-testing, an any-hit shader is run to sample an opacity texture.
- Once the original ray generation thread finishes, it progressively updates the radiance stored in a floating-point accumulation texture.

Because we will be using bindless techniques to access our resources, we can completely forego local root signatures in favor of a single global root signature used for the entire call to `DispatchRays`. This global root signature will contain the following:

- An SRV descriptor table containing our entire shader-visible descriptor heap (with overlapping ranges for each resource type).
- A root SRV descriptor for our scene's acceleration structure.
- A UAV descriptor table containing a descriptor for our accumulation texture.[2]
- A root CBV descriptor containing global constants filled out by the CPU just before the call to `DispatchRays`.
- Another root CBV containing constants that contain application settings whose values come from a runtime UI controls.

Because we do not need any local root signatures, we can completely skip adding any `D3D12_LOCAL_ROOT_SIGNATURE` subobjects to our state object, as well as the `D3D12_SUBOBJECT_TO_EXPORTS_ASSOCIATION` subjects for associating a local root signature with an export. This also means that each shader record in our shader tables only needs to contain the shader identifier, as we have no parameters for a local root signature.

When we described the basic functionality of our path tracer, we mentioned how it needs to compute lighting response every time a path intersects with geometry in the scene. To do this, we need to properly reconstruct the surface attributes at the point of intersection. For attributes such as normals and UV coordinates that are derived from per-vertex attributes, our hit shaders will

[2] Bindless UAVs can also be used on a device that supports `D3D12_RESOURCE_BINDING_TIER_3`, which is currently the case for all DXR-capable GPUs. However, the example path tracer referenced by this chapter only utilizes bindless techniques for SRVs.

need to access the vertex and index buffers in order to find the three relevant triangle vertices and interpolate their data. To facilitate this, we will create a `StructuredBuffer` whose elements are defined by the following struct:

```
struct GeometryInfo
{
    uint VertexOffset;  // Can alternatively be descriptor indices
                        // to unique vertex/index buffer SRVs
    uint IndexOffset;
    uint MaterialIndex; // Assumes a single global buffer, could also
                        // have a descriptor index for the buffer
                        // and then another index into that buffer
    uint PadTo16Bytes;  // For performance reasons, not required
};
```

Our buffer will be created with one of these elements for every `D3D12_RAYTRACING_GEOMETRY_DESC` that was provided when building the scene acceleration structure, which in our case corresponds to a single mesh that was present in the original source scene. By doing it this way, we can conveniently index into this buffer using the `GeometryIndex()` intrinsic that is available to our hit shaders.[3]

To complement our `GeometryInfo` buffer, we will also build another structured buffer containing one entry for every unique material in the scene. The elements of this buffer will be defined by the following struct:

```
struct Material
{
    uint Albedo;
    uint Normal;
    uint Roughness;
    uint Metallic;
    uint Opacity;
    uint Emissive;
};
```

Each `uint` in this struct is the index of a descriptor in our global descriptor heap, which allows hit shaders to access those textures through the SRV descriptor table in the global root signature. Note that this assumes a rather uniform material model, where all materials use the same set of textures and don't require any additional parameters. However, it would be trivial to add additional values to this struct if necessary, provided that those values are common to all materials. It would also be straightforward for a material to indicate that it did not need to sample a particular texture by providing a known invalid index (such as `uint32_t(-1)`). The hit shaders could then check for this and branch over the texture sample if necessary. A more complex approach might involve having the `Material` struct instead provide

[3] `GeometryIndex()` is a new intrinsic that was added for DXR 1.1.

the index of a CBV descriptor, whose layout would be interpreted by the individual hit shaders. Alternatively, a set of heterogeneous data could be packed into a single `ByteAddressBuffer`, where again it would be up to the hit shader to interpret and load the data appropriately based on the material's requirements.

With our shader tables, state object, and geometry/material buffers built, we can now call `DispatchRays()` to launch many threads of our ray generation program. As we mentioned earlier, our simple path tracer will work by launching one thread for every pixel in the screen. Each of these threads then starts out by using its associated pixel coordinate to compute a camera ray according to a standard perspective projection, which effectively serves as a simple pinhole camera model. Once we've computed the appropriate ray direction, we can then call `TraceRay()` to trace a ray from the camera's position into our scene. This ray's payload contains a `float3` that will be set to the computed radiance being reflected or emitted toward the camera, which can then be written into an output texture through a `RWTexture2D`. Our path tracer is progressive, which means that it will compute `N` radiance samples per pixel every frame (with `N` defaulting to 1) and update the output accumulation texture with these samples. This allows the path tracer to work its way toward a final converged image by doing a portion of the work every frame, thus remaining interactive.

In order to compute the outgoing radiance at each hit point (or *vertex* in path tracer terminology), our closest-hit shader needs to compute the surface attributes at the hit point and then compute the appropriate lighting response. Our first step is to use our `GeometryInfo` buffer to fetch the relevant data for the particular mesh that was intersected by the ray:

```
const uint geoInfoBufferIndex = GlobalCB.GeoInfoBufferIndex;
StructuredBuffer<GeometryInfo> geoInfoBuffer;
geoInfoBuffer = GeometryInfoBuffers[geoInfoBufferIndex];
const GeometryInfo geoInfo = geoInfoBuffer[GeometryIndex()];
```

Once we have the relevant data for the mesh that was hit, we can then begin to reconstruct the surface attributes by interpolating the per-vertex attributes according to the barycentric coordinates of the ray/triangle intersection:

```
MeshVertex GetHitSurface(
    in HitAttributes attr, in GeometryInfo geoInfo)
{
    float3 barycentrics;
    barycentrics.x = 1 - attr.barycentrics.x - attr.barycentrics.y;
    barycentrics.y = attr.barycentrics.x;
    barycentrics.z = attr.barycentrics.y;

```

```
    StructuredBuffer<MeshVertex> vertexBuffer;
    vertexBuffer = VertexBuffers[GlobalCB.VertexBufferIndex];
    Buffer<uint> indexBuffer;
    indexBuffer = BufferUintTable[GlobalCB.IndexBufferIndex];

    const uint primId = PrimitiveIndex();
    const uint id0 = indexBuffer[primId * 3 + geoInfo.IndexOffset + 0];
    const uint id1 = indexBuffer[primId * 3 + geoInfo.IndexOffset + 1];
    const uint id2 = indexBuffer[primId * 3 + geoInfo.IndexOffset + 2];

    const MeshVertex vtx0 = vertexBuffer[id0 + geoInfo.VertexOffset];
    const MeshVertex vtx1 = vertexBuffer[id1 + geoInfo.VertexOffset];
    const MeshVertex vtx2 = vertexBuffer[id2 + geoInfo.VertexOffset];

    return BarycentricLerp(vertex0, vertex1, vertex2, barycentrics);
}
```

For surface attributes defined by textures, we must fetch the appropriate material definition and use it to sample our standard set of textures.

```
StructuredBuffer<Material> materialBuffer;
materialBuffer = MaterialBuffers[GlobalCB.MaterialBufferIndex];
const Material material = materialBuffer[geoInfo.MaterialIndex];

Texture2D albedoMap    = Tex2DTable[material.Albedo];
Texture2D normalMap    = Tex2DTable[material.Normal];
Texture2D roughnessMap = Tex2DTable[material.Roughness];
Texture2D metallicMap  = Tex2DTable[material.Metallic];
Texture2D emissiveMap  = Tex2DTable[material.Emissive];

// Sample textures and compute final surface attributes.
```

Note how we are able to effectively sample from any arbitrary set of textures here while using only a single hit shader and no local root signatures! This use case perfectly demonstrates the power and flexibility of bindless techniques: all resources are at our disposal from within a shader, and we are able to store handles to those resource in any arbitrary data structure that we would like to use.

With the surface attributes and material properties for the hit point loaded into local variables, we can finally run our path tracing algorithm to compute an incremental portion of the radiance that will either be emitted from the surface or reflected off the surface in the direction of the incoming ray. The full algorithm is as follows:

- Get the set of material textures and sample them using vertex UV coordinates to build surface and shading parameters.
- Account for emission from the surface.
- Sample direct lighting from the sun and cast a shadow ray.

- Sample direct lighting from local light sources and cast a shadow ray for each.
- Choose to sample a diffuse or specular BRDF.
  - If diffuse, choose a random cosine-weighted sample on the hemisphere surrounding the surface normal.
  - If specular, choose a random sample from the distribution of visible microfacet normals and reflect a ray off of that.
- Recursively evaluate the incoming radiance in the sample direction.
- Terminate when the desired max path length is reached.
  - Alternatively, use Russian roulette to terminate rays with low throughput.

As for our miss shaders, we merely need to sample our procedural sky model in order to account for the radiance that it provides the scene. We can do this by sampling a cube map texture that contains a cache of radiance values for each world-space direction, which we can obtain by passing its descriptor index to the shader through a global constant buffer. One exception is for primary rays that were cast directly from the camera: for this case we also want to sample the emitted radiance from our procedural sun, which allows the sun to be visible in our rendered images. We skip this for secondary rays cast from surfaces, as we directly importance-sample the sun to compute its direct lighting contribution.

```
[shader("miss")]
void MissShader(inout PrimaryPayload payload)
{
    const float3 rayDir = WorldRayDirection();

    TextureCube skyTexture = TexCubeTable[RayTraceCB.SkyTextureIdx];
    float3 radiance = 0.0f;
    if(AppSettings.EnableSky)
        radiance = skyTexture.SampleLevel(SkySampler, rayDir, 0).xyz;

    if(payload.PathLength == 1)
    {
        float cosSunAngle = dot(rayDir, RayTraceCB.SunDirectionWS);
        if(cosSunAngle >= RayTraceCB.CosSunAngularRadius)
            radiance = RayTraceCB.SunRenderColor;
    }

    payload.Radiance = radiance;
}
```

To support foliage that utilizes alpha testing, we provide an any-hit shader that samples an opacity map to determine if the ray intersection should be discarded. We obtain this opacity map from the material data using the same method that we utilized for the hit shader:

```
[shader("anyhit")]
void AnyHitShader(inout PrimaryPayload payload, in HitAttributes attr)
{
    const uint geoInfoBufferIndex = GlobalCB.GeoInfoBufferIndex;
    StructuredBuffer<GeometryInfo> geoInfoBuffer;
    geoInfoBuffer = GeometryInfoBuffers[geoInfoBufferIndex];

    const GeometryInfo geoInfo = geoInfoBuffer[GeometryIndex()];
    const MeshVertex hitSurface = GetHitSurface(attr, geoInfo);

    StructuredBuffer<Material> materialBuffer;
    materialBuffer = MaterialBuffers[GlobalCB.MaterialBufferIndex];
    const Material material = materialBuffer[geoInfo.MaterialIndex];

    // Standard alpha testing
    Texture2D opacityMap = Tex2DTable[material.Opacity];
    if(opacityMap.SampleLevel(MeshSampler, hitSurface.UV, 0).x < 0.35f)
        IgnoreHit();
}
```

17.5 PRACTICAL IMPLICATIONS OF USING BINDLESS TECHNIQUES

As this chapter has demonstrated, there are many tangible benefits to utilizing bindless techniques in a D3D12 renderer. This is true regardless of whether a renderer makes use of ray tracing through DXR or sticks to more traditional rasterization-based techniques. However, there are also several downsides and practical implications of which one should be aware before broadly adopting a bindless approach.

17.5.1 MINIMUM HARDWARE REQUIREMENTS

Before using bindless techniques at all, it's important to know the minimum hardware requirements for using them. As of the time this chapter was written, all DXR-capable hardware has support for `D3D12_RESOURCE_BINDING_TIER_3`, which is capable of utilizing bindless techniques for all shader-accessible resource types. This means that bindless can be utilized with DXR without concern that the underlying hardware or driver will not have support for dynamic indexing into an unbounded SRV, UAV, or sampler table. However, it may be important to consider what functionality is supported on older hardware if writing a general D3D12 rendering engine that supports older generations of video cards.

While the descriptor heap and table abstractions exposed by D3D12 suggest that they only run on the sort of hardware that can read descriptors from memory (and are therefore capable of using arbitrary descriptors in a shader program), in reality the API was carefully designed to allow for older D3D11-era hardware to be supported through compliant drivers. This class of hardware falls under the Tier 1 resource binding tier, where the device reports a value of `D3D12_RESOURCE_BINDING_TIER_1` for the `ResourceBindingTier` member of the `D3D12_FEATURE_DATA_D3D12_OPTIONS` structure. Devices with Tier 1 resource binding limit the size of bound SRV descriptor tables to the maximum number of SRVs supported by D3D11 (128), which is far less than the number of textures and/or buffers that most engines will have loaded at any given time. Therefore, we would generally consider Tier 1 hardware to be incapable of using bindless techniques, at least outside of certain special-case scenarios. As of the time this chapter was written, GPUs based on NVIDIA's Fermi architecture (GTX 400 and 500 series) as well as Intel Gen 7.5 (Haswell) and Gen 8 (Broadwell) architecture will report Tier 1 for resource binding [10].

Hardware that falls under the category of Tier 2 resource binding are largely free of restrictions when it comes to accessing SRV descriptors: these devices allow SRV descriptor tables to span the maximum heap size, which is one million. This limit is sufficient for including all shader-readable resources in typical engines, so we would consider this class of hardware to be bindless-capable for SRVs. Tier 2 hardware can also bind a full heap of sampler descriptors (which is capped at 2048), which we would also consider to be sufficient for bindless sampler access in typical scenarios. However, there are still limitations for this tier. Tier 2 has a maximum size of 14 for CBV descriptor tables and a maximum size of 64 for UAV descriptor tables. In addition, this tier imposes restrictions that require that all bound CBV/UAV descriptor tables contain valid, initialized descriptors. As of the time this chapter was written, only GPUs based on NVIDIA's Kepler architecture (GTX 600 and 700 series) will report Tier 2 for resource binding.[4]

The most recent graphics hardware will report Tier 3 for resource binding, which removes the Tier 2 limitations on the number of UAV and CBV descriptors that can be bound simultaneously in a descriptor table. Tier 3 allows for the full heap to be bound for CBV and UAV tables and also removes the requirement that these tables contain only valid, initialized descriptors. With these restrictions removed, we can say that Tier 3 is bindless-capable for

[4] NVIDIA hardware based on the Maxwell (GTX 900 series) and Pascal (GTX 1000 series) initially reported Tier 2 for resource binding in their earliest D3D12 drivers, but later drivers add full support for Tier 3.

all shader-accessible resources: Shader Resource Views, Unordered Access Views, Constant Buffer Views, and Sampler States. This provides us with maximum freedom and flexibility in terms of providing our shaders with access to our resources. As of the time this chapter was written, GPUs based on NVIDIA's Maxwell (GTX 900 series), Pascal (GTX 1000 series), Volta (Titan V), Turing (RTX 2000 series and GTX 1600 series), and Ampere (RTX 3000 series) architectures as well as Intel Gen 9 (Skylake), Gen 9.5 (Kaby Lake), Gen 11 (Ice Lake), and Gen 12 (Tiger Lake) report Tier 3 capabilities for resource binding. Tier 3 is also reported by all D3D12-capable AMD hardware, which includes GCN 1 through 5 as well as newer RDNA-based GPUs.

17.5.2 VALIDATION AND DEBUGGING TOOLS

D3D12 includes an optional debugging and validation layer that can be invaluable for development. When enabled, this layer can report issues relating to incorrect API usage and invalid descriptors used by shaders, as well as incorrect or missing resource transition barriers. Unfortunately, these kinds of validations become much more difficult to perform for an application that uses bindless techniques for accessing resources. With a more traditional binding setup where all required descriptors are provided in a contiguous descriptor table, a validation layer can track CPU-side API calls in order to determine the set of descriptors that will be accessed within a particular draw or dispatch call. This information can then be used to ensure that these descriptors all point to valid resources and that these resources have been previously transitioned to the appropriate state.

When an application deploys bindless techniques, the set of descriptors accessed by a draw or dispatch is no longer visible through CPU-side API calls. Rather, they can only be determined by following the exact flow of execution that happens on the GPU in order to compute the descriptor index that is used to ultimately access a particular descriptor. To address these scenarios, D3D12 also has a special GPU-based validation layer (abbreviated as GBV). When GBV is enabled, which is done by calling `ID3D12Debug1::SetEnableGPUBasedValidation()`, the validation layer will patch the compiled shader binaries in order to insert instructions that log the accessed descriptors to a hidden buffer. The contents of this buffer can later be inspected, and the information contained can be used to validate descriptor contents and resource states. This functionality makes it a bona fide requirement for development of bindless renderers. Unfortunately, the shader patching process can add considerable time to the pipeline state object (PSO) creation process, and the patched shaders also cause additional

performance overhead on the GPU. Therefore, it is generally recommended to only use it when necessary.

GPU debugging tools such as RenderDoc, PIX, and NVIDIA Nsight Graphics can also be invaluable for solving bugs during development. Unfortunately, they suffer from the same issues that validation layers experience when dealing with bindless applications: it is no longer possible to determine the set of accessed descriptors/resources for a draw or dispatch purely through interception of CPU-side API calls. Without that information, a tool might report *all* of the resources that were bound through the global descriptor tables, which would likely be the entire descriptor heap for a bindless application. Fortunately, PIX has a solution to this problem: when running analysis on a capture, it can utilize techniques similar to those used by the GPU-based validation layer to determine the set of accessed descriptors for a draw or dispatch. This allows the tool to display those resources appropriately in the State view, which can provide a similar debugging experience to traditional binding. However, it is important to keep in mind that this set of accessed resources can potentially grow very large for cases where descriptor indices are nonuniform within a draw or dispatch.

17.5.3 CRASHES AND UNDEFINED BEHAVIOR

While bindless techniques offer great freedom and flexibility, they also include the potential for new categories of bugs. Though accessing an invalid descriptor is certainly possible with traditional binding methods, it's generally easier to avoid because descriptor tables are populated on the CPU. With bindless resources we can instead expose an entire descriptor heap to our shaders, which provides more opportunities for bugs to cause an invalid descriptor to be accessed. In a sense these descriptor indices are very similar to using pointers in C or C++ code on the CPU, as they are both quite flexible but also provide ample opportunities to access something invalid.

When an invalid descriptor is accessed by a shader, the results are undefined. The shader program might end up reading garbage resource data that ultimately manifests as graphical artifacts, or it might cause the D3D12 device to enter a *removed* state that occurs after a fatal error is detected. Because the behavior is unpredictable, it is recommended to leverage both the D3D validation layers as well as in-engine validation as much possible. Similar to the concept of a `NULL` pointer, it is possible to reserve a known value such as `uint32_t(-1)` to be used as an "invalid" descriptor index. This value can then be used when initializing structures containing descriptor indices, and

debugging code can be written to assert that the index is valid before passing the data off to the GPU908. It is also possible to write GPU-side shader code that validates descriptor indices before usage, although it can be much more complicated to report the results of that validation to the programmer.

To aid in debugging situations where a device removal occurs, it is also recommended to implement a GPU crash detection and reporting system within a renderer. Ideally such a system can tell you which particular command the GPU was executing when it crashed or hung, and potentially provide some additional information as to the specific reason for device removal. D3D12 has built-in functionality known as *Device Removed Extended Data* (DRED) [3] that can facilitate the gathering of this information. When enabled, DRED will automatically insert "breadcrumbs" into command lists that can be read back following a crash in order to determine which commands actually completed on the GPU. The low-level mechanisms utilized by DRED can also be triggered manually for applications that want to implement their own custom system for tracking GPU progress. In addition, NVIDIA offers their own proprietary Aftermath library [4] that can provide additional details if the crash occurs on an NVIDIA GPU. In particular, it can provide source-level information about the shader code that was executing when device removal occurred, provided the shaders were compiled with appropriate debug information.

17.6 UPCOMING D3D12 FEATURES

In Section 17.3, which discussed implementing bindless resources in D3D12, we described the specific steps required for adding the overlapping descriptor ranges and unbounded arrays of HLSL resource types that are needed for allowing our shaders to have global access of all resources. This process is rather clunky and limiting because it adds quite a bit of boilerplate and doesn't scale well with the effectively unlimited permutations of the `StructuredBuffer` and `ConstantBuffer` template types. Though the current setup is workable, it certainly makes one wish for future changes that could simplify the process of implementing bindless resources.

Fortunately, Microsoft has released a set of HLSL additions as part of Shader Model 6.6 [8]. These new features include a dramatically simplified syntax for globally accessing the descriptors within the bound descriptor heaps via the new `ResourceDescriptorHeap` and `SamplerDescriptorHeap` objects:

```
1 StructuredBuffer<Material> materialBuffer;
2 materialBuffer = ResourceDescriptorHeap[GlobalCB.MaterialBufferIndex];
3 const Material material = materialBuffer[geoInfo.MaterialIndex];
```

```
4
5  Texture2D albedoMap    = ResourceDescriptorHeap[material.Albedo];
6  Texture2D normalMap    = ResourceDescriptorHeap[material.Normal];
7  Texture2D roughnessMap = ResourceDescriptorHeap[material.Roughness];
8  Texture2D metallicMap  = ResourceDescriptorHeap[material.Metallic];
9  Texture2D emissiveMap  = ResourceDescriptorHeap[material.Emissive];
10
11 SamplerState matSampler = SamplerDescriptorHeap[material.SamplerIdx];
```

This new syntax completely removes the need for declaring unbound arrays of HLSL resource types and also makes it unnecessary to add any descriptor table ranges and root parameters to the root signature. The only requirement is that the root signature be created with a new `D3D12_ROOT_SIGNATURE_FLAG_CBV_SRV_UAV_HEAP_DIRECTLY_INDEXED` flag that is available in the latest version of the D3D12 headers.

Taken together, this new functionality greatly reduces the boilerplate and friction encountered when using bindless techniques with earlier shader models. Consequently, it is expected that the new syntax will become the dominant approach once the new shader model is widely available to developers and end users, and that the older syntax will eventually fall out of use. However, we do not feel that this fundamentally changes the overall benefits and drawbacks of using bindless techniques, therefore it will still be important to consider the trade-offs presented earlier in the chapter before deciding whether or not to adopt it.

17.7 CONCLUSION

In this chapter we demonstrated how bindless techniques can be used in the context of DirectX Raytracing and discussed the various benefits and practical implications of utilizing bindless resources for this purpose. We hope that the reasons for choosing bindless are quite clear, especially when it comes to future rendering techniques that make use of DXR 1.1 inline tracing. Bindless techniques are here to stay and are likely to be a core feature of future APIs and graphics hardware. For current development, programmers are free to determine whether to adopt them completely throughout their renderers or to selectively make use them for specific scenarios.

For a complete example of a simple DXR path tracer that makes full use of bindless techniques for accessing read-only resources, please consult the DXRPathTracer GitHub repository [9]. Full source code is included for both the application and shader programs, and the project is set up to be easily compiled and run on a DXR-capable Windows 10 PC with Visual Studio 2019 installed.

REFERENCES

[1] Barrett, S. Sparse Virtual Textures. https://silverspaceship.com/src/svt/, 2008. Accessed September 6, 2020.

[2] Bolz, J. OpenGL bindless extensions. https://developer.download.nvidia.com/opengl/tutorials/bindless_graphics.pdf, 2009. Accessed May 16, 2020.

[3] Kristiansen, B. New in D3D12—DRED helps developers diagnose GPU faults. https://devblogs.microsoft.com/directx/dred/, January 24, 2019. Accessed August 23, 2020.

[4] NVIDIA. NVIDIA Nsight Aftermath SDK. https://developer.nvidia.com/nsight-aftermath, 2020. Accessed February 2, 2021.

[5] NVIDIA. Unreal engine sun temple scene. https://developer.nvidia.com/ue4-sun-temple. Accessed June 10, 2021.

[6] Pettineo, M. Bindless texturing for deferred rendering and decals. https://therealmjp.github.io/posts/bindless-texturing-for-deferred-rendering-and-decals/, March 25, 2016. Accessed May 16, 2020.

[7] Pettineo, M. DXR Path Tracer. https://github.com/TheRealMJP/DXRPathTracer, 2018. Accessed May 2, 2020.

[8] Roth, G. Announcing HLSL Shader Model 6.6. https://devblogs.microsoft.com/directx/hlsl-shader-model-6-6/, April 20, 2021. Accessed April 24, 2021.

[9] TheRealMJP. DXRPathTracer. https://github.com/TheRealMJP/DXRPathTracer. Accessed June 10, 2021.

[10] Wikipedia. Feature levels in Direct3D. https://en.wikipedia.org/wiki/Feature_levels_in_Direct3D, 2015. Accessed August 5, 2020.

[11] Wyman, C. and Marrs, A. Introduction to DirectX Raytracing. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 21–47. Apress, 2019.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 18

WEBRAYS: RAY TRACING ON THE WEB

Nick Vitsas,[1,2] Anastasios Gkaravelis,[1,2] Andreas A. Vasilakis,[1,2] and Georgios Papaioannou[1]
[1] Athens University of Economics and Business
[2] Phasmatic

ABSTRACT

This chapter introduces WebRays, a GPU-accelerated ray intersection engine for the World Wide Web. It aims to offer a flexible and easy-to-use programming interface for robust and high-performance ray intersection tests on modern browsers. We cover design considerations, best practices, and usage examples for several ray tracing tasks.

18.1 INTRODUCTION

Traditionally, ray tracing has been employed in rendering algorithms for production and interactive visualization running on high-end desktop or server platforms, relying on dedicated, native, and close-to-the-metal libraries for optimal performance [2, 5]. Nowadays, the Web is the most ubiquitous, collaborative, convenient, and platform-independent conveyor of visual information. The only requirement for accessing visual content online is a web browser on an Internet-enabled device, dispensing with the dependence on additional software.

The potential of the Web has driven Khronos to release the WebGL specifications in order to standardize a substantial portion of graphics acceleration capabilities on the majority of new consumer devices and browsers. As a result, frameworks like ThreeJS and BabylonJS emerged, enabling various visualization applications that take advantage of the Web as a platform [4]. However, these solutions have been explicitly designed for widespread commodity rasterization-based graphics, before ray tracing was popular. There is currently no functionality exposed that accommodates client-side GPU-accelerated ray tracing on this platform.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_18

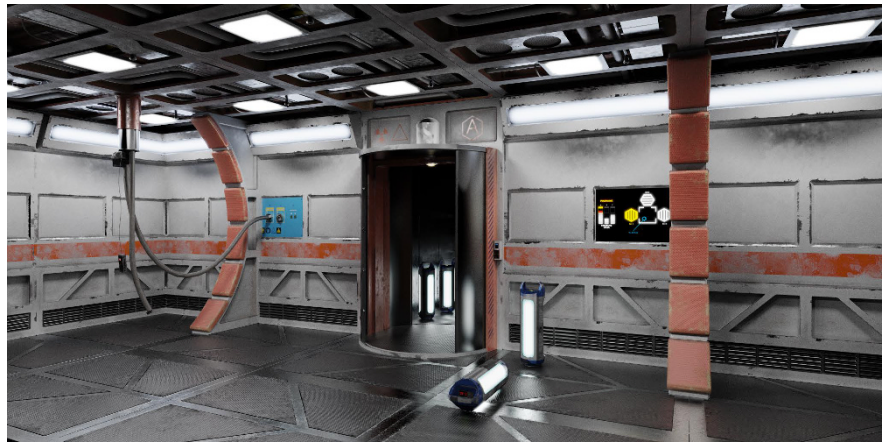


Figure 18-1. *The Space Station scene rendered using a unidirectional path tracer implemented in WebRays (trace depth 4).*

We developed WebRays to fill this gap, by aiming for an as-thin-as-possible abstraction over any underlying graphics, or potentially compute, application programming interface (API). Following the successful design of modern ray tracing engines, WebRays exposes an easy-to-use and explicit API with lightweight support for acceleration data structures, to enable ray/triangle intersection functionality in Web-based applications. It is by design not intended to implement a specific image synthesis pipeline. Instead, it offers a framework for ray tracing support that can either act as a core API for implementing diverse rendering pipelines and algorithms or as a complementary toolkit to harmoniously coexist with and enhance rasterization-based graphics solutions (see the example in Figure 18-1). This is achieved by allowing access to ray tracing functionality both with stand-alone intersection procedures and individual ray tracing function calls from within a standard shader program.

A prototype version of the API was successfully utilized for the implementation of the Rayground platform [7]. Rayground is an open, cross-platform, online integrated development sandbox and educational resource for fast prototyping and interactive demonstration of ray tracing algorithms. It offers a programming experience similar to a standard, GPU-accelerated ray tracing pipeline that allows customization of the basic ray tracing stages directly in the browser.

In this chapter, we cover the system architecture and programming interface design as well as provide representative usage examples via code snippets for a gentle introduction to WebRays.

18.2 FRAMEWORK ARCHITECTURE

Modern APIs that support ray intersection acceleration, such as DXR and OptiX [2, 5], are designed to give explicit control to developers. Users are given as much freedom as possible over resource management and control flow. WebRays sets a similar goal.

18.2.1 DESIGN GOALS

We want to bring ray tracing functionality to modern browsers, with no specific conditions or exceptions and irrespective of platform. We strive for independence from a particular specialized or low-level graphics API. Users refer to engine resources using opaque handles, and information about internal storage types is only communicated to the user in order to help achieve optimal performance. Having often been on the user end of similar frameworks, we recognized the need for a modern API that simultaneously supports two different approaches to ray tracing:

- *Wavefront:* A simple, general-purpose execution model for tracing arbitrary ray batches in bulk.

- *Megakernel:* A programming interface for ray tracing within a GPU shader.

In wavefront ray tracing, the entire process is divided into small, discrete phases, corresponding to specific kernels, which are executed successively and process data in parallel (Figure 18-2, left). On the other hand, in the

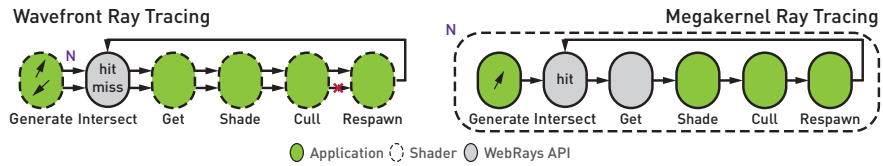


Figure 18-2. *Wavefront and megakernel ray tracing conceptual models. Left: ray tracing consists of simple parallel steps. Note that all stages (except Intersect) can be executed on either host side or device side. Right: A monolithic kernel is responsible for the entire ray tracing process.*

megakernel approach, a single kernel is launched. All operations including intersections, visibility determination, shading, etc. take place within this single kernel (Figure 18-2, right). This allows user code to closely follow the underlying algorithm and can therefore be considered more programmer-friendly. However, given the nature of modern streaming processors, it can lead to serious occupancy and divergence issues as the potential execution paths and access patterns may vary significantly.

Each approach operates under different design considerations, and we wanted to properly support both. In WebRays, this is completely up to application logic, and the user can also combine the two strategies. To this end, the API offers both a *host-side* interface implemented in JavaScript and a *device-side* interface implemented in the backend's shading language (GLSL). Performance-sensitive parts of the library are written in C++ and compiled to WebAssembly.

WebRays is not intended as a stand-alone pipeline for image synthesis. It does not introduce features such as specific shading models and material properties, nor does it enforce a particular image synthesis algorithm. In the same spirit, with WebRays you do not write kernels that respond to specific events. The developer is free to implement their own pipeline, which of course comes at the cost of more user-defined "glue" code. With this generic, yet complete, ray tracing support, one can even build wrapper APIs on top of WebRays to conform to popular ray tracing frameworks. Finally, capturing the typical needs of practical rendering implementations, it currently supports triangle primitives.

18.2.2 HOST-SIDE API

The JavaScript API is accessible through the standard web development workflow. Working with this host-side API is very similar to most ray intersection engines. To start, users submit triangle meshes to build acceleration data structures over their geometric data. These acceleration data structures are built on the CPU and later uploaded to the GPU for fast intersections. The API uses handles to refer to these structures on both the host and the device sides. Users "submit" rays for intersection by allocating and populating appropriately formatted ray buffers. Similarly, intersection or occlusion results are returned to the user through appropriately formatted intersection buffers. These allocations are handled by the user to allow for full control over the application's GPU memory management. To facilitate a

wavefront approach, the host-side API offers functions that take a batch of rays, identify ray/geometry intersections in parallel, and finally store the results in properly formatted buffers. Nothing needs to leave the GPU in this process, and the results are ready to be consumed in a shader or fetched back to the CPU.

18.2.3 DEVICE-SIDE API

The device-side API enables users to write megakernels and hybrid rendering solutions. The biggest challenge is to be able to provide access to geometric data and ray tracing functionality in a flexible and unobtrusive manner. Like early OpenGL, current programmable graphics pipelines for the Web are configured with string-based shaders provided in source form. In WebRays, the shader code that constitutes the device-side API for ray intersection functionality is a simple string, automatically generated by the engine. Developers can acquire this string via an API call and prepend it to their own source code.

The attached code gives access to in-shader intersection routines and helper functions for accessing geometric properties of intersected objects. We did not want to offer yet another shader abstraction layer as this would not fit well with existing graphics applications that want to add ray intersection functionality. All calls and variables are appropriately isolated in their own namespace, in order to avoid clashes with users' code.

The WebRays intersection engine allocates GPU resources for internal data structures, which need to be bound to the user's program for the device-side API to function properly. These resources are commonly passed from the host to the device through shader binding locations. The API offers a function that returns a binding table containing the name, type, and value of the resource that needs to be bound to the shader before execution.

WebRays is currently implemented on top of WebGL 2.0. The remainder of the text exposes several WebGL-specific implementation details and best practices. Familiarity with any OpenGL version will greatly help the reader.

18.2.4 ENGINE CORE

Device-side computation in WebGL 2.0 is available via plain fragment shaders. Also, memory allocation and sharing must be handled through standard textures. When memory needs to be exchanged between WebRays and the

rest of the application, e.g., passing a ray buffer for intersection and receiving the results, it is passed using appropriately formatted textures. Thankfully, the support of multiple render targets in WebGL 2.0 makes this process quite streamlined.

The intersection engine and the application achieve interoperability by both operating on the same WebGL context. In order to help users manage their GPU resources, WebRays is transparent about its memory requirements, layouts, and bindings. These design decisions help the intersection engine work closely with the application and allow for maximum performance by keeping ray tracing resources resident on the GPU every step of the way.

18.2.5 ACCELERATION DATA STRUCTURES

Geometry loading from external sources is left to application code, similar to other modern APIs [2, 5]. WebRays is optimized for triangle meshes, which constitute the most common geometric representation for 3D models. The API supports two types of acceleration data structures (ADS): top-level (scene-wide) acceleration structure (TLAS) and bottom-level (per-object) acceleration structure (BLAS). A TLAS can contain one or more BLASs. Instancing is also supported by inserting the same BLAS into the TLAS multiple times with different transformation matrices.

The internals of the WebRays data structures are not exposed to the user. The library internally allocates and deallocates GPU resources for triangle data. We employ bounding volume hierarchies (BVHs) as acceleration data structures, as they have proven their worth in terms of balanced performance and construction cost. Specifically, we use a custom variant of a wide BVH [8], which has shown good performance for incoherent rays.

18.3 PROGRAMMING WITH WEBRAYS

The basic control flow of a WebRays application is outlined in Figure 18-3. The general steps are *setup*, *init*, *update*, and *execute*, all of which are determined by the context of the intended execution model and algorithm.

- *Setup:* The application gets a handle to the WebRays host-side API functions (Section 18.3.1).
- *Init:* The ADSs based on the scene's geometry are created (Section 18.3.2). In case of wavefront ray intersections (Section 18.3.5), ray and intersection buffers are also allocated and filled accordingly (Section 18.3.3).

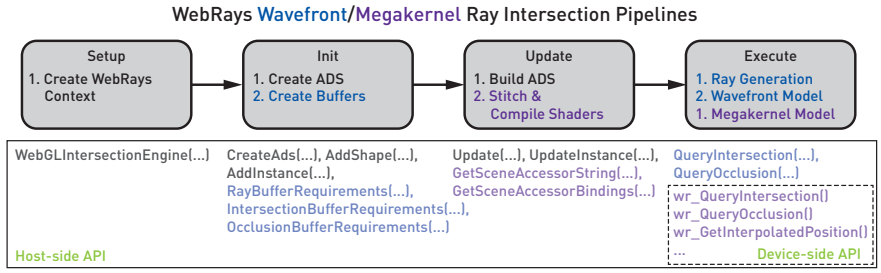


Figure 18-3. *Top: basic WebRays application control flow for wavefront and megakernel ray tracing. Steps in black are common for both pipelines. Bottom: a summary of the basic functions exposed by the WebRays API and how these map to each application stage.*

- *Update:* The ADSs are actually built (or updated), and data is synchronized between CPU and GPU memory (Section 18.3.2). In the case of megakernel ray tracing, the dynamically created WebRays device-side API code must be prepended to the user-provided fragment shader, before compilation.

- *Execute:* Rays are generated (Section 18.3.4) and their intersections handled either within a viewport-filling rectangle draw pass (wavefront rendering) or directly within the user fragment shader in the megakernel approach (Section 18.3.6). After execution has finished, the resulting intersection data can be used by the application in a multi-pass manner by repeating the cycle.

18.3.1 SETUP

For WebRays to function in the HTML document, users specify a canvas object that will be enabled with WebGL capabilities at runtime. The WebRays API is accessible via a single JavaScript file, loaded as usual:

```
<body>
  <canvas id="canvas" width="640" height="480"></canvas>
</body>
<script src="webrays.js"></script>
```

Because all access to device-side rendering is handled via WebGL, one also needs to access the WebGL context in one's own script:

```
var gl = document.querySelector("#canvas").getContext("webgl2");
```

The intersection engine context is initialized by providing the newly created WebGL context:

```
var wr = new WebRays.WebGLIntersectionEngine(gl);
```

From this point on, users can start using both the `gl` and the `wr` objects to perform rasterization and ray tracing, respectively.

18.3.2 POPULATING THE ACCELERATION DATA STRUCTURES

Before tracing any rays, a bottom-level acceleration structure must first be constructed for the mesh geometry. In the following code, we assume that the mesh has been loaded using an external library with the corresponding mesh class providing access to the vertex, normal, and texture coordinate buffers, along with their respective strides:

```
function build_blas_ads(mesh) {
  let blas  = wr.CreateAds({ type: "BLAS" });
  let shape = wr.AddShape(blas,
                          mesh.vertex_data, mesh.vertex_stride,
                          mesh.normal_data, mesh.normal_stride,
                          mesh.uv_data    , mesh.uv_stride,
                          mesh.face_data);
  return [blas, shape];
}
```

Each vertex position and normal vector is a `float[3]` and each texture coordinate pair a `float[2]`. For the index buffer, the API expects an `int[4]`. The `xyz` components contain the offsets to each of the triangle's vertices within the attribute buffers. The `w` component is user-provided and is always available during intersections. This can be used to store application-specific data like material indices, texture indices, and more.

The `blas` variable is an opaque handle that can later be passed to host-side or device-side WebRays API functions in order to refer to this specific structure. The user can add multiple shapes to a single BLAS. The returned `shape` variable is a handle that is used by the engine to identify the specific geometry group within the BLAS.

A TLAS is created similarly to a BLAS, but the user adds *instances* of existing BLASs to the TLAS via their handles:

```
let blases = [];
let tlas   = wr.CreateAds({ type: "TLAS" });
for (mesh in meshes) {
    let [blas, shape] = build_blas_ads(mesh);
    let instance      = wr.AddInstance(tlas, blas, mesh.transform);
    blases.push(blas);
}
```

An instance's transformation can later be updated using the returned instance handle with the following code:

```
1 wr.UpdateInstance(tlas, instance, transform);
```

Up to this point, none of the provided scene geometry resources have been submitted to the GPU. The user must call the following function in order to commit and finalize the created structures:

```
1 let flags = wr.Update();
```

The returned flags indicate whether the device-side accessor code string or the bindings were affected by the update operation. It is advised to call this function every frame, as it does not incur an additional cost when no actual update is required and it enables the programmer to react on a significant change.

For example, if the returned flags indicate a change in the device-side API, the user is expected to get the new device-side API code as well as the updated bindings using the following snippet:

```
1 let accessor_code = wr.GetSceneAccessorString();
2 let bindings      = wr.GetSceneAccessorBindings();
```

The accessor code is a plain string that needs to be prepended to the user's shader code before compilation. Bindings are mostly relevant at program invocation time during rendering.

18.3.3 RAY AND INTERSECTION BUFFERS

The ray structure declaration that WebRays uses internally resembles the one described in the *Ray Tracing Gems* chapter "What Is a Ray?" [6]. Intersection data use a packed representation and are not intended to be directly read by the user. However, they can be passed to API functions within the shader in order to get intersection-specific attributes. Occlusion data simply indicate if any geometric object was found between the user-defined ray extents.

Ray buffers have memory requirements for two `vec4` entries. Users are required to fill two buffers corresponding to a `vec4(origin, tmin)` and a `vec4(direction, tmax)` in order to submit rays for intersection. Intersection and occlusion buffers use integer buffers. Because ray tracing commonly requires high-precision arithmetic operations, the API provides helper functions to give a hint to the user about which is the optimal storage and internal texture format.

```
let dimensions    = [width, height];
let ray_req       = wr.RayBufferRequirements(dimensions);
let intersect_req = wr.IntersectionBufferRequirements(dimensions);
let occlusion_req = wr.OcclusionBufferRequirements(dimensions);
let origins       = tex2d_alloc(ray_req);
let directions    = tex2d_alloc(ray_req);
let intersections = tex2d_alloc(intersect_req);
let occlusions    = tex2d_alloc(occlusion_req);
```

The preceding code allocates two `vec4` buffers for storing ray information and two integer ones for storing intersection and occlusion results. The returned requirement structure contains the type of the buffer along with the expected format and dimensions. The engine supports both 1D and 2D buffers, depending on the dimensions of the passed array. Note that 2D buffers fit naturally to image synthesis. For example, in the current WebGL implementation, the most common buffer type is backed by an appropriately formatted 2D texture. Thus, the `tex2d_alloc` call is an application helper function for allocating such buffers in WebGL.

18.3.4 RAY GENERATION

Ray buffers can be populated either in the host-side JavaScript code or by utilizing a GPU shader. Each design choice depends on the performance characteristics of the application. The simplest and most efficient method to populate ray buffers is by using the native graphics API. Because WebGL does not have access to compute shaders, launching a shader for ray generation and/or handling of intersection data on the device side is simply performed via a fragment shader over a viewport-filling rectangle in a specific viewport matching the ray batch size. Although one can fill ray data and read back intersections in the host-side code via a read pixels operation, it is advantageous to always perform such operations on the GPU.

For example in WebGL, the textured-backed ray buffers can be attached as render targets in a framebuffer object, and ray properties can be written using a fragment shader. This way, setting the ray direction, origin, and valid ray interval is trivial:

```
origin_OUT    = vec4(origin,    tmin);
direction_OUT = vec4(direction, tmax);
```

The newly populated textures can then be passed to WebRays as ray buffers for intersection or used as input in any stage of the application's rendering pipeline. It is important to note that intermediate storage of rays is not a requirement in order to perform intersections. Using the device-side

intersection API, a ray can be generated, intersected, and consumed within a single shader. A typical use case is that of shadow rays (Section 18.4.3), where a new ray can be sampled and queried directly within the shader with the result being accessible immediately.

18.3.5 HOST-SIDE INTERSECTIONS

WebRays offers host-side API functions for ray intersection and occlusion queries. Intersection queries return closest-hit information encoded in an `ivec4`. Occlusion queries are useful for binary visibility queries. As expected, they are faster than regular intersection queries, due to their any-hit termination criterion.

```
1 let rays = [origins, directions];
2 wr.QueryIntersection(ads, rays, intersections, dimensions);
3 wr.QueryOcclusion   (ads, rays, occlusions,    dimensions);
```

The `ads` that the API expects is the same handle that is created on the host side during acceleration data structure creation (Section 18.3.2). `origins` and `directions` are the properly formatted and populated ray buffer textures, whereas `intersections` and `occlusions` are appropriately formatted intersection buffer textures (Section 18.3.3) that will receive intersection and occlusion results, respectively.

Because in essence all these buffers are basic textures, the programmer can pass them to shaders as regular uniform sampler variables. This way, users can bind the already-filled intersection texture and apply application-specific operations in parallel, on the GPU. This approach completely decouples intersection operations from application logic, giving users full control of how they manage ray queries and process the results. The same holds for the consumption of intersection results.

18.3.6 DEVICE-SIDE INTERSECTIONS

The device-side API enables intersection queries within the shader as well as access to geometric properties of intersected objects. For intersection queries, WebRays provides two function variants, `wr_QueryIntersection` and `wr_QueryOcclusion`. These take as a parameter the handle of a previously created acceleration data structure `ads`. `wr_QueryOcclusion` returns the result of a visibility test, whereas `wr_QueryIntersection` provides the closest intersection point encoded in a single `ivec4`. This value can be subsequently passed to other intersection-specific API accessor functions to obtain

interpolated position, normal, texture parameters, barycentric coordinates, etc.

The following shader code, which implements a basic renderer using ray casting, demonstrates the usage of those two functions. Primary ray hits can either be calculated in the shader (lines 8–10) or obtained from a previous separate shader invocation (line 12). The `ads` uniform can be treated as a single `int` and passed to the shader with the appropriate `glUniform` variant.

```
1  uniform int        ads;
2  uniform sampler2D  origins;
3  uniform sampler2D  directions;
4  uniform isampler2D intersections;
5  ...
6  ivec2 coords    = ivec2(gl_FragCoord.xy);
7  // Case 1: Query primary ray intersections.
8  vec4  origin    = texelFetch(origins, coords, 0);
9  vec4  direction = texelFetch(directions, coords, 0);
10 ivec4 hit       = wr_QueryIntersection(ads, origin, direction, tmax);
11 // Case 2: Or obtain already calculated primary ray hits.
12 ivec4 hit       = texelFetch(intersections, coords, 0);
13 ...
14 // Miss
15 if (!wr_IsValidIntersection(hit)) {
16   color_OUT = ...
17   return;
18 }
19 // Intersect
20 ivec4 face        = wr_GetFace(ads, hit);
21 vec3  geom_normal = wr_GetGeomNormal(ads, hit);
22 vec3  normal      = wr_GetInterpolatedNormal(ads, hit);
23 vec3  position    = wr_GetInterpolatedPosition(ads, hit);
24 ...
25 // Shade intersected point
26 color_OUT = ...
```

18.4 USE CASES

This section describes how a number of fundamental and modern applications of ray tracing to image synthesis can be implemented with WebRays. We demonstrate how the versatile design and high-performance implementation of WebRays can be used in pure ray tracing implementations (Sections 18.4.1 and 18.4.2) as well as hybrid rendering (Section 18.4.3). Timings across several devices are provided for all experiments in Figure 18-4, to give a feeling of the expected performance. The complete source code of the WebRays examples is provided in the accompanying repository of the book. The source code aims to get developers started with WebRays, moving from trivially simple to moderately complex ray tracing examples. Furthermore, we briefly describe Rayground (Section 18.4.4), an interactive authoring platform for ray tracing algorithms based on WebRays.

Scene (triangles) / Device	 (35k)	 (143k)	 (143k)	 (109k)	 (51k)	 (4.6k)	 (123k)
nVidia RTX 2080	21	5.3	51	12.2	6.2	4.3	44
nVidia GTX 1060†	50	5.1	140	35.0	19.5	3.0	147
nVidia GTX 970	60	5.8	188	41.0	25.0	4.5	193
AMD RX 580	54	6.0	200	37.0	9.0	5.0	192
Intel HD 630†	290	34.2	626	131.0	93.0	14.0	618

Figure 18-4. *Expected performance on a variety of desktop and laptop† GPU devices. Timings show the total frame time, in milliseconds, of each use case, including shading, intersections, and post-processing. Scenes are rendered at a native* 1024×768 *resolution.*

18.4.1 AMBIENT OCCLUSION

Ambient occlusion is a non-physically-accurate illumination technique, highly popularized in the games industry due to its relative simplicity and efficiency. Using ray tracing, it can be estimated by stochastically sampling the visibility of the shaded point over a hemisphere centered at its normal vector.

The following code demonstrates how WebRays visibility queries can be used to estimate ambient occlusion in a fragment shader. The `sample_count` corresponds to the number of directions selected by importance sampling, and `dmax` the near-field extent. The `position` and `normal` of the shaded point come from the previous ray casting example of Section 18.3.6, but they can be derived by other means, such as a forward rasterization pass (Section 18.4.3). The shaded result using the Fireplace scene and a near-field extent of 1.5 m is shown in Figure 18-5, left.

```
...
float occlusion  = 0.0;
for (int s = 0; s < sample_count; s++) {
  vec3 direction = CosineSampleHemisphere(normal);
  bool occluded  = wr_QueryOcclusion(ads, position, direction, dmax);
  occlusion     += occluded ? 1.0 : 0.0;
}
occlusion /= float(sample_count);
// Ambient lighting visualization
color_OUT  = vec4(vec3(1.0 - occlusion), 1.0);
```

18.4.2 PATH TRACING

Path tracing is an elegant method that estimates the integral of the rendering equation using Monte Carlo simulation. It produces accurate photorealistic images because all kinds of light transport paths are supported and sampled.

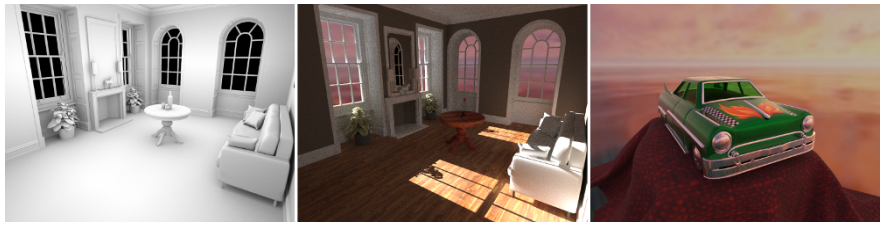


Figure 18-5. *Monte Carlo integration in WebRays. Left: the Fireplace scene using ray traced near-field ambient occlusion. Middle and right: a unidirectional path tracer (trace depth 4) with next event estimation used for the Fireplace scene (middle) and the ToyCar model (right).*

Using WebRays, a path tracer can be developed in either a wavefront or a megakernel manner. Due to the lack of bindless resources in WebGL, materials should be packed into a shared buffer and textures should be packed into a texture atlas, in order to minimize texture bindings. In the following code, a unidirectional path tracer with next event estimation is implemented using a wavefront approach, where rays are generated using a pinhole camera and intersections are resolved using the primitive and material ID of the intersected primitive.

```
[origins, dirs] = Generate(camera); // Get primary rays.
while(depth < depthmax) {
  // Ray intersection test
  wr.QueryIntersection (ads, [origins, dirs], hits, dimensions);
  // Get new shadow and outgoing rays.
  [origins, occ_dirs, dirs] = ResolveIntersections(hits);
  // Shadow rays test
  wr.QueryOcclusion(ads, [origins, occ_dirs], occs, dimensions);
  // Compute local illumination.
  ResolveDirectIllumination(hits, occs);
  depth++; // Increase path length.
}
```

The `ResolveIntersections` call is responsible for populating the next outgoing ray buffer and the shadow ray buffer and is listed in the following code. This can be done efficiently by launching a single 2D kernel. Using multiple render targets, the kernel is responsible for populating the ray texture buffers. The last two textures correspond to the shadow rays and the outgoing rays leaving the intersection point. The first texture stores the ray origin, which is the same for both rays.

```
vec3  wo     = -direction_IN.xyz;
vec3  origin = wr_GetInterpolatedPosition(ads, hit);
vec3  normal = wr_GetGeomNormal(ads, hit);

```

```
float light_distance;
vec3  shadow_ray, wi;
LightSample(origin, /*out*/shadow_ray, /*out*/light_distance);
BxDF_Sample(origin, wo, /*out*/wi, /*out*/scattering_pdf);

direction_OUT        = vec4(wi, tmax);
shadow_direction_OUT = vec4(shadow_ray, light_distance - RAY_EPSILON);

origin     += normal * RAY_EPSILON;
origin_OUT  = vec4(origin, RAY_EPSILON);
```

After executing this kernel, the textures are forwarded to `QueryIntersection` and `QueryOcclusion` respectively. Direct illumination is computed in `ResolveDirectIllumination` based on the results produced from `QueryOcclusion` and the bidirectional scattering distribution function `BxDF` of the hit point.

```
vec3 origin    = wr_GetInterpolatedPosition(ads, hit);
vec3 Li        = LightEval(origin, light, wi, light_pdf, light_dis);
...
vec3 Ld        = vec3(0.0);
int  occlusion = texelFetch(occlusions, coords, 0).r;
if(occlusion == 0) {
  vec3  BxDF   = BxDF_Eval(origin, wo, wi);
  Ld          += throughput * BxDF * NdL * Li / light_pdf;
}
color_OUT = vec4(Ld, 0.0);
```

Figures 18-1 and 18-5 (middle and right) show how this efficient WebRays path tracing implementation can be used to compute global illumination images for interior and outdoor scenes.

18.4.3 HYBRID RENDERING

Rendering on the Web is usually performed using a rasterization-based pipeline via either a deferred or a forward rendering approach. With WebRays, ray tracing becomes available to both desktop and mobile browsers and can be used to replace or complement some components of the typical rasterization pipeline, enhancing the quality of the rendered image.

AMBIENT OCCLUSION

In real-time applications, either ambient occlusion is precomputed, for static scenes, or near-field ambient occlusion is computed in real time, using screen-space information stored in the G-buffer. Screen-space techniques, due to their view-dependent nature, fail to capture occlusion from objects that are offscreen or occluded by other geometric objects from the camera's point of view. Ray tracing can be utilized to compute accurate ambient occlusion

without these drawbacks. The shaded point's position and normal can be reconstructed from the G-buffer, and ambient occlusion can be estimated as described in Section 18.4.1.

SHADOWS

Ray-traced shadows provide accurate and crisp boundaries and can work in complex lighting conditions such as arbitrarily shaped light sources. They offer a superior method for shadow calculations than the prevalent method for computing shadows in real-time graphics that uses shadow maps [1].

In a hybrid renderer, the shaded point position can be reconstructed from the depth buffer, populated during the G-buffer rasterization or a depth prepass step. Then, any number of occlusion rays can be cast toward the light sources and checked for visibility.

```
// Shadow ray computation
vec3  origin     = ReconstructPositionFromDepthBuffer();
float distance;
vec3  direction;
LightSample(origin, /*out*/direction, /*out*/distance);
// Shadow visibility test
bool  occluded   = wr_QueryOcclusion(ads, origin, direction, distance);
float visibility = occluded ? 0.0 : 1.0;
```

Occlusion queries are very fast as they terminate on the first encountered primitive intersection. Alpha-tested geometry would require a more complex hit kernel, where transparency of the hit point on the primitive should be taken into account. Soft shadows can be easily computed by sampling directions toward the light source surface area. Figure 18-6, left, presents soft shadows of thin geometry produced from a distant light source, whereas in the middle of the figure, colored soft shadows produced from a textured area light are demonstrated.

REFLECTION AND REFRACTION

Complex light paths resulting from reflected and transmitted light are very difficult to compute using a rasterization pipeline. Ray tracing offers a robust way to handle such difficult phenomena.

In a hybrid rendering pipeline, rays are spawned from the reconstructed position from the depth buffer toward a random direction inside the reflection or transmission lobe defined by the material properties and traced into the scene. Each hit point is then evaluated and shaded, and a new ray is spawned and traced into the scene. This recursive procedure is usually performed up

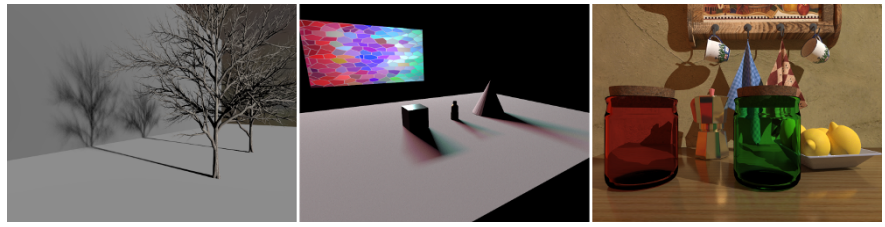


Figure 18-6. *Hybrid rendering in WebRays. Left and middle: accurate soft shadows in the Tree and Mosaic scenes. Right: reflection and transmission events (trace depth 6) captured in the Kitchen Table scene.*

to a maximum number of iterations `depthmax` based on the number of light paths that the specific scenario requires. An example using WebRays is presented in the following code, where the recursive procedure is performed inside a single shader and rays are traced using the in-shader API call `wr_QueryIntersection`. Each hit point is evaluated and a new ray is spawned, or the procedure terminates in the event of a miss. Accurate reflection and transmission results from the above shader are presented in Figure 18-6, right.

```
// Geometry information computation
vec3 position = ReconstructPositionFromDepthBuffer();
vec3 normal   = ReconstructNormalFromGBuffer();
BxDF bxdf     = ReconstructMaterialFromGBuffer();
vec3 wo       = normalize(u_camera_pos - position);

// Compute new ray based on the material.
vec3 wi, throughput = vec3(1.0), color = vec3(0.0);
Sample_bxdf(bxdf, normal, wo, /*out*/wi, /*out*/throughput);

while(depth < depthmax) {
  ivec4 hit  = wr_QueryIntersection(ads, position, wi, tmax);
  if (!wr_IsValidIntersection(hit)) { // Miss
    color   += throughput * EvaluateEnvironmentalMap(wi);
    break;
  }
  else { // Surface hit
    position = wr_GetInterpolatedPosition(ads, hit);
    normal   = wr_GetInterpolatedNormal(ads, hit);
    bxdf     = GetMaterialFromIntersection(hit);
    wo       = -wi;

    vec3 Ld  = EvaluateDirectLight(position, normal, bxdf, wo);
    color   += throughput * Ld;

    Sample_bxdf(bxdf, normal, wo, /*out*/wi, /*out*/throughput);
  }
  depth++;
}
color_OUT = color;
```

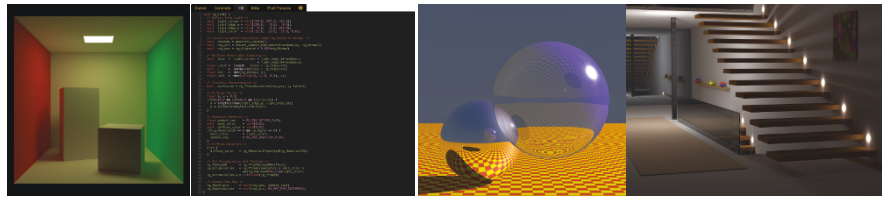


Figure 18-7. *Left: the Rayground interface, with the preview window and the shader editor. Right: two representative ray traced projects created online on this platform: Whitted ray tracing and stochastic path tracing.*

18.4.4 RAY TRACING PROTOTYPING PLATFORM

WebRays has been deployed at the core of the Rayground platform, which is hosted at https://www.rayground.com. The Rayground pipeline offers a high-level framework for easy and rapid prototyping of ray tracing algorithms. More specifically, it exposes one declarative stage and four programmable ones. In the declarative stage, users describe their scene using simple shape primitives and material properties. The scene is then submitted to WebRays in order to build and traverse the acceleration data structure. The four programmable stages are the standard *generate*, *hit*, *miss*, and *post-process* events. The graphical user interface consists of two discrete sections, the preview window and the shader editor. Visual feedback is interactively provided in the preview canvas, and the user performs live source code modifications as shown in Figure 18-7.

18.5 CONCLUSIONS AND FUTURE WORK

We have presented WebRays, a GPU-accelerated ray intersection framework able to provide photorealistic 3D graphics on the web. The versatile design of WebRays allows it to adapt to a broad range of application requirements, ranging from simple intersection queries to complex visualization tasks.

Admittedly, the low-level nature of the API requires a good amount of boilerplate code from the user. In the future, we intend to provide wrapper abstraction APIs that model more constrained yet popular programmable ray tracing pipelines. Keeping an eye toward future advances on Web-based accelerated graphics, we plan to offer a WebGPU backend implementation of WebRays, as soon as the latter becomes available to browsers, in order to provide a standard way to express ray tracing ubiquitously, so it ultimately benefits the entire 3D online graphics industry.

ACKNOWLEDGMENTS

This work was funded by the Epic MegaGrants grant program from Epic Games Inc. The ToyCar model, created by Guido Odendahl, was downloaded from Khronos Group repository. The San Miguel (whose Tree model was used) and Fireplace scenes were downloaded from McGuire's Computer Graphics Archive [3]. The remaining scenes were created by the authors.

REFERENCES

[1] Eisemann, E., Schwarz, M., Assarsson, U., and Wimmer, M. *Real-Time Shadows*. A K Peters/CRC Press, 1st edition, 2011. DOI: 10.1201/b11030.

[2] Haines, E. and Akenine-Möller, T. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress, 2019. DOI: 10.1007/978-1-4842-4427-2.

[3] McGuire, M. Computer graphics archive. https://casual-effects.com/data, 2017.

[4] Mwalongo, F., Krone, M., Reina, G., and Ertl, T. State-of-the-art report in Web-based visualization. *Computer Graphics Forum*, 35(3):553–575, 2016. DOI: 10.1111/cgf.12929.

[5] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. OptiX: A general purpose ray tracing engine. *ACM Transations on Graphics*, 29(4):66:1–66:13, July 2010. DOI: 10.1145/1778765.1778803.

[6] Shirley, P., Wald, I., Akenine-Möller, T., and Haines, E. What is a ray? In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 15–19. Apress, 2019.

[7] Vitsas, N., Gkaravelis, A., Vasilakis, A.-A., Vardis, K., and Papaioannou, G. Rayground: An online educational tool for ray tracing. In *Eurographics 2020—Education Papers*, pages 1–8, 2020. DOI: 10.2312/eged.20201027.

[8] Ylitie, H., Karras, T., and Laine, S. Efficient incoherent ray traversal on GPUs through compressed wide BVHs. In *Proceedings of High Performance Graphics*, HPG '17, 4:1–4:13, 2017. DOI: 10.1145/3105762.3105773.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 19

VISUALIZING AND COMMUNICATING ERRORS IN RENDERED IMAGES

Pontus Andersson, Jim Nilsson, and Tomas Akenine-Möller
NVIDIA

ABSTRACT

In rendering research and development, it is important to have a formalized way of visualizing and communicating how and where errors occur when rendering with a given algorithm. Such evaluation is often done by comparing the test image to a ground-truth reference image. We present a tool for doing this for both low and high dynamic range images. Our tool is based on a perception-motivated error metric, which computes an error map image. For high dynamic range images, it also computes a visualization of the exposures that may generate large errors.

19.1 INTRODUCTION

Rendering comparisons are often done using one of three methods. One may, for example, display the results of two algorithms next to each other and a ground-truth reference next to them. Another method, called *pooling*, shows tables containing single values, acquired using image error measures, corresponding to the error between the algorithms' outputs and the reference. There are several issues with both of these approaches, however. Showing images next to each other makes it much more difficult to spot differences that could be rather large, and it may also hide smaller errors, such as chrominance differences, which can be hard for a human observer to find. One problem with condensing the errors in an entire image down to a single value is that it will discard information about where the errors are located, which may be crucial knowledge during algorithm comparisons. For example, the same, condensed error could be given both for a test image with several fireflies or for a test image suffering small color shifts—using the pooled error, we cannot tell the difference.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_19

Another presentation option is to show a full error map, with the same resolution as the reference and test images, in which each pixel value indicates the error between the two images in that pixel. When using an error map, we gain the possibility to pinpoint where in the test image errors are present, which in turn can aid us in deciding their causes and impact on the visual experience. Several works have used error maps to compare and present their results. However, many have used error metrics whose outputs are not necessarily indicative of the error an observer would perceive when shown the images. Instead, the error metrics most commonly used are simple difference computations, such as the relative mean squared error (relMSE) or the symmetric mean absolute percentage error (SMAPE), which are not based on human perception and often do not agree with it. Other commonly used error metrics also exhibit nonintuitive error maps [3]. The SSIM [18] metric, though popular, has similar issues, as has been further explained elsewhere [14].

In this chapter, we will give an overview of an alternative metric called ꟻLIP [3, 4] and present a tool which is based on it. ꟻLIP is inspired by perception research and the qualities of the human visual system. It outputs a per-pixel error map that indicates where an observer would perceive errors, and the magnitude those errors would be perceived to have, as the observer *alternates* between the reference and test images on top of each other. ꟻLIP was designed with this alternation in focus, as it is the viewing protocol most commonly used by rendering practitioners. Originally, ꟻLIP was limited to images with low dynamic range (LDR) [3]. It was later extended to handle high dynamic range (HDR) images [4] as well. Because HDR images are exposure compensated and tone mapped before they are shown to the viewer on an LDR display, it is the error in the resulting LDR image that is of interest. An important part of this, however, is the choice of exposure. As illustrated in Figure 19-1, different errors can be seen under different exposures. The HDR version of ꟻLIP takes this into account by considering several exposures during its evaluation. After giving a brief overview of the LDR and HDR versions of the metric, we introduce a tool that leverages both versions and that a developer may use to visualize and communicate the perceived errors in rendered images. In the final section, we use the results of ꟻLIP to evaluate the quality of different algorithm configurations and investigate how it relates to their performance.

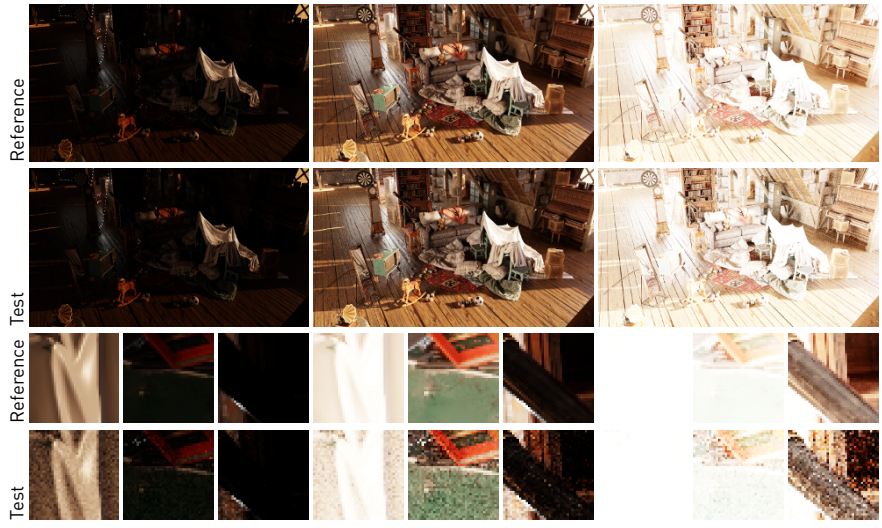


Figure 19-1. *A reference image rendered at two million samples per pixel (spp) and a test image at 256 spp of the Attic scene are shown under three different exposures. The tool presented in this chapter allows the user to compute, visualize, and communicate the error an observer would perceive between the LDR test and reference images when alternating between them. In the zoom-ins, we see that the errors have different magnitude in the different test images. Some errors are largest in the left image, some in the middle one, and some in the right image. While large in one image, the error could also be unnoticeable in another. When working with HDR images, the exposure compensation and tone mapping carried out before the images are displayed is sometimes unknown, meaning that the HDR image could be displayed as any of the LDR images presented in this figure. Thus, examining the error in an HDR test image requires the user to compare the probable LDR versions of it to the corresponding LDR reference images—a tedious and time-consuming task if done manually, but which is automated by our tool.*

19.2 ꟻLIP

We base our work on the low dynamic range and high dynamic range versions of ꟻLIP, which is a perception-motivated error metric targeting rendered images. The original metric, LDR-ꟻLIP [3], acts on LDR images, while the second version, HDR-ꟻLIP [4], was made to handle HDR images. Our tool, presented in Section 19.3, chooses the option that corresponds to the dynamic range of the input images. Next, we briefly introduce LDR- and HDR-ꟻLIP.

19.2.1 LDR-ꟻLIP

LDR-ꟻLIP [3] takes an LDR reference image and an LDR test image and produces an error map together with *pooled values*, which are numbers that, in different ways, aim to describe the overall error between the two images.

In particular, LDR-ꟻLIP approximates the difference an observer would perceive when alternating between the test and reference images—a viewing protocol often used in rendering algorithm development. The algorithm consists of a color pipeline and a feature pipeline, both of which consider the observer's distance from the display and the display's size, and combines the results into an error map.

The color pipeline first applies spatial filters, based on contrast sensitivity functions, to the images. This removes high-frequency detail that we cannot perceive at a certain distance from the display. The images are then transformed into a perceptually uniform color space, i.e., a space where distances between colors correspond better with the perceived distance than in, e.g., RGB space. In LDR-ꟻLIP, the perceptually uniform color space of choice was the simple L*a*b* space [7]. A small adjustment, to account for the Hunt effect [8], is then applied. The Hunt effect predicts that color differences appear larger at higher luminances than lower ones. The color distance is then measured using a metric [1] that works well also for large color differences, which are often present in rendered images. This distance is then mapped into a color difference in [0, 1] in a way that compresses large color differences, for which it is hard to tell which difference is larger, to a smaller range and that extends the range for smaller differences.

The feature pipeline aims at comparing the edges and points (collectively called features) in the images. Because the human visual system is sensitive to edges and because points, e.g., fireflies, often occur in rendered images, comparing the feature content between the reference and test images is important to assess the quality of the test image. In the feature pipeline, features are first detected in both the reference and test images. A feature difference is then computed between the results and is mapped to [0, 1], similar to the color difference. The per-pixel error, called ΔE, is then

$$\Delta E = \left(\Delta E_c\right)^{1-\Delta E_f}, \tag{19.1}$$

where ΔE_c is the color difference and ΔE_f is the feature difference.

In some applications, providing and examining error maps for each image pair is not feasible. In such cases, some level of pooling, or data reduction, is necessary. Both the LDR and HDR versions of ꟻLIP provide such pooling alternatives. As a first step, the user may choose to reduce the final LDR- or HDR-ꟻLIP error map into a weighted histogram, where each bin has been weighted by the error in the center of the bin. This weighting will make larger

errors more pronounced, which helps the user spot the presence of artifacts such as fireflies, information that is otherwise easily clouded by the large number of lower errors in the image. The histogram plots also contain indicators for statistical measures of the error map, including the mean, weighted median, minimum, and maximum error. These values are always output by our tool. If only one value is to be recorded per image pair, the mean is a relatively good option, as discussed in Section 19.6. We refer the reader to Section 19.4 for an example where pooling is used.

19.2.2 HDR-ꟻLIP

While low dynamic range images have color components with integer values in [0, 255], or those values remapped to 256 values in [0, 1], high dynamic range images can have any values. We have, however, limited our work to values that are nonnegative. Recently, ꟻLIP was extended to handle HDR images [4]. HDR images are often exposure compensated, by multiplying the image with a factor 2^c, where c is the exposure compensation value, and then tone mapped before they are shown on an LDR display. As noted in Section 19.1, errors in HDR images that are visible using a certain exposure compensation value may be invisible at another exposure compensation value, and vice versa. In many cases, such as in real-time rendered games, the exposure compensation value may be determined arithmetically on the fly, which possibly leads to a wide range of exposures. This was considered by Munkberg et al. [12], who proposed an idea where the HDR images were exposure compensated using several exposures within a certain range and then the mean squared error (MSE) was accumulated over the results and turned into a peak signal to noise (PSNR) value.

The idea of computing errors over an exposure range was refined by Andersson et al. [4], as follows. First, the exposure compensation start and stop values, c_{start} and c_{stop}, are computed automatically from the reference HDR image. Second, both exposure compensation and tone mapping are taken into account. Third, instead of MSE, the perception-motivated LDR-ꟻLIP measure is used on each pair of LDR images. Finally, instead of taking the per-pixel average, the maximum of all LDR-ꟻLIP values per pixel is recorded, as that is the largest error an observer might see when shown the HDR image on an LDR display. This resulted in a more reliable error measure for HDR images compared to mPSNR [12], RMAE, SMAPE [17], relMSE [16], and HDR-VDP-2 [11].

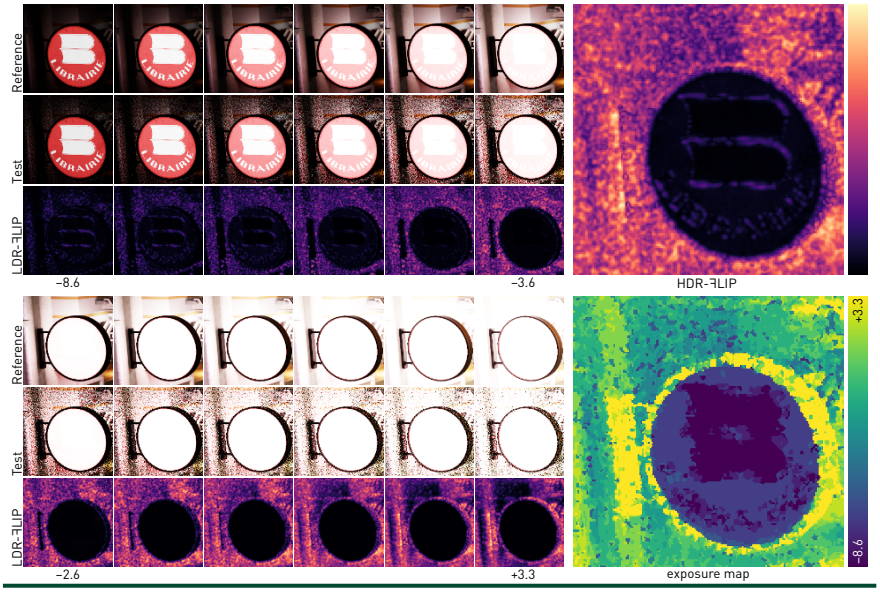


Figure 19-2. *An illustration of how HDR-FLIP works using a crop of an image from the BistroNight scene. A reference HDR image is compared against a test HDR image, where the latter was path traced with 16 samples per pixel. The top two rows of small images show the reference (top) and the test (middle) images, after exposure compensation and tone mapping. The respective LDR-FLIP images of these two are shown in the third row. The bottom three rows contain the same types of images, but for the second half of the exposure range. The entire exposure range goes from* $c_{\text{start}} = -8.6$ *to* $c_{\text{stop}} = 3.3$*. The HDR-FLIP error map (top right) contains the per-pixel maximum error of all LDR-FLIP images. The exposure map (bottom right) visualizes from which exposure the maximum error originates, where dark is the shortest exposure, within the used exposure range, and bright is the longest. This means, for example, that the maximum errors on the Librairie sign comes from the lower exposures. The HDR-FLIP error map reveals, however, that those errors are small.*

Figure 19-2 illustrates the power of HDR-FLIP. For these results, we have used the FLIP default of 67 pixels per degree in the calculations. Unless otherwise stated, this will be what is used for each FLIP result presented in this chapter. Per default, the error maps produced by FLIP use the *magma* color map [10], shown in the top right of the figure, to indicate different error levels. It goes from almost black for low errors to a "hot" color for large errors. These are properties that make the interpretation of the error map intuitive and easy as, e.g., zero error maps to black. This, together with the fact that it is perceptual in the sense that each increment in error maps to the same increment in perceived color difference and that it is friendly to those who are color blind, is why the magma map was chosen. The exposure map

too uses a certain color map, called *viridis* [10]. It shares the perceptual and color-blind-friendliness of magma, and its luminance increases over the map. Unlike magma, viridis is not black for the lowest value. This is beneficial for the interpretation of the exposure map as the lowest value in an exposure map could be anything, not just zero as was the case for the error maps. Looking back at the results in Figure 19-2, it is noteworthy that there are errors in the LDR-ꟻLIP image with the lowest exposure and errors in the LDR-ꟻLIP image with the highest exposure that both end up in the composite HDR-ꟻLIP image, as is revealed by the exposure map.

Besides helping with visualizing the error in HDR images, HDR-ꟻLIP also saves time for developers. Indeed, for HDR images, a developer would need to alternate between the multitude of possible LDR versions of the reference and test images, each pair generated using a different exposure compensation value, in order to, for example, be convinced that the errors are sufficiently small. HDR-ꟻLIP gathers this information into a single error map, which makes for a significantly faster analysis.

19.3 THE TOOL

With this chapter, we provide a command line tool, written in C++, CUDA, and Python, that computes and outputs ꟻLIP information according to the user's preferences. The tool, code, and instructions are publicly available at https://github.com/NVlabs/flip. In this section, we explain what the tool does and in the next we provide and discuss example use cases.

The minimum required inputs to the ꟻLIP tool are one reference image and one test image. For LDR images, the default output from ꟻLIP is the full error map together with the mean, minimum, maximum, weighted median, weighted first quartile, and weighted third quartile error. As explained in Section 19.6, the mean ꟻLIP value is what is preferably communicated if the metric is reduced to a single number. For HDR images, the default output is the same as for LDR images, with the addition of the automatically calculated exposure range. Furthermore, using a command line argument, the tool can output the weighted histogram of the ꟻLIP error map, as seen in the top of Figure 19-5 in the next section. The tool can also produce an overlapping histogram, as seen at the bottom of Figure 19-5. Options also include using a logarithmic scale on the y-axis, and setting a maximum value on the y-axis for easier comparison between different test outcomes.

Some options affect the results directly. For both LDR and HDR input, the user may input the observer's distance from the monitor (in meters) and the monitor's width (in meters and pixels). With these, the number of pixels per degree can be computed, which will be used in the ꟻLIP calculations. Alternatively, one may input the number of pixels per degree directly. In the case of HDR input, the user may choose to input either one or both of the start and stop exposures, as well as the total number of exposures that will be used in the calculation. Manually choosing the start and stop exposures can be beneficial when they are known, which may eliminate unnecessary calculations and error indications. Per default, the start and stop exposures are computed automatically as described in the HDR-ꟻLIP paper [4]. For HDR, the user may also choose the underlying tone mapper of HDR-ꟻLIP. By default, an ACES approximation [13] is used, and it is the tone mapper used for all results in this chapter.

Finally, the tool provides some extra output options. The user may choose to output the ꟻLIP error map in grayscale or magma (default), or choose not to output an error map at all. In the case of HDR images, outputting the exposure map is optional. It is also possible to output all intermediate (exposure compensated and tone mapped) LDR images resulting from the HDR input images. The full set of options and instructions are available at https://github.com/NVlabs/flip.

19.4 EXAMPLE USAGE AND OUTPUT

In this section, we will demonstrate the usage of our tool. First, we will study an LDR example, where most focus will be on the LDR-ꟻLIP error map. After that, two algorithms that generate HDR output will be compared using HDR-ꟻLIP's pooled variants, weighted histograms, error maps, and exposure maps.

Figure 19-3 shows a reference and a test image [6] of the Buddha scene. Recall that the LDR-ꟻLIP error map represents the error an observer sees while alternating between the reference and test images under the given viewing conditions. From the error map alone, we can see where those errors are located and how large they are. For example, we see that there is a large error on the upper right arm of the statue (to the left in the image), as well as in the background. This is easily confirmed by looking at the LDR images, or even better by flipping between the two. The error map also indicates some error on the foot of the statue that is generally smaller than the error on the

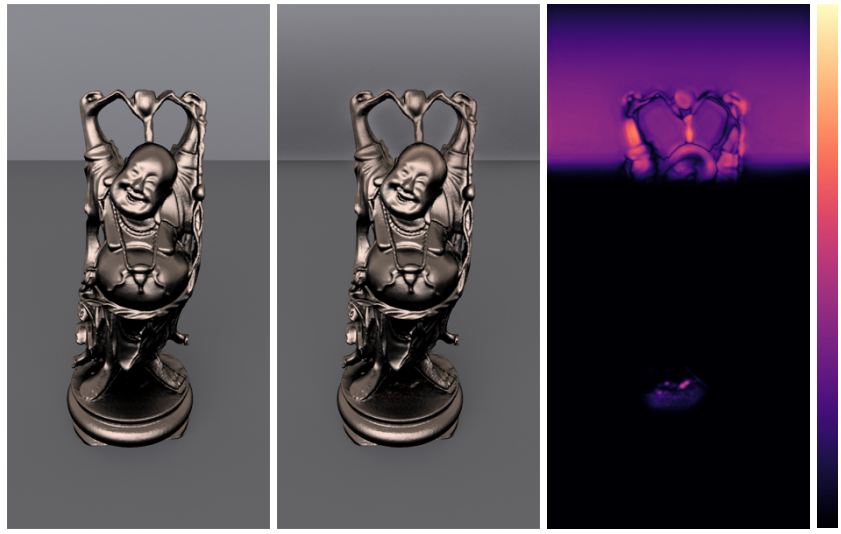


Figure 19-3. *Reference and test images of the Buddha scene, together with the LDR-ꟻLIP map.*

arm and the background, which can be confirmed by examining the LDR images. With this example, we see how LDR-ꟻLIP can help us to quickly get an overview of the magnitude and location of the differences an observer would perceive when alternating between two images and how this can help communicate the differences without explicitly alternating between them.

Next, we will use the tool to examine the error in HDR images. As opposed to the LDR-ꟻLIP error map, the HDR-ꟻLIP result cannot be said to represent the error an observer would see while alternating between the images, as we never present the HDR images, but only exposure compensated and tone mapped versions of them. This is under the assumption that we use LDR displays. The HDR-ꟻLIP error map instead represents the largest error per pixel that an observer would see while alternating between pairs of equally exposure compensated and tone mapped reference and test images. The exposure range and tone mapper used for this are chosen such that the HDR-ꟻLIP error map is computed using a large portion of the probable LDR stimuli that the two HDR images could generate. We note that the exposure range and tone mapper can be manually chosen as well (Section 19.3), in case any of them is known.

In the next example, we have rendered the ZeroDay scene with two different direct illumination algorithms, simply denoted A and B, respectively, and we

will use our tool to compare the two. The reference and the two test images, for a certain exposure compensation factor, are shown in Figure 19-4. We have marked some difficult areas for the two algorithms, in green for algorithm A and in blue for B. The region marked in white seems to be handled well by both algorithms. First, we will look at the pooled error values, specifically the mean HDR-ꟻLIP error for A and B. Using three decimal digits, we have that the mean error for A is 0.524, while the mean error for B is 0.525, which implies that the two algorithms perform similarly overall. Looking at Figure 19-4, however, we see that the algorithms generate very different outputs.

In Figure 19-5, we present the weighted histograms of the HDR-ꟻLIP error maps for A and B as well as a plot where the histograms are overlapped, which facilitates a visual comparison of the two. The histograms reveal that the errors are slightly differently distributed, where A produces more errors in the mid-high range, while the image generated by B has more errors in the mid and top range, with the latter implying artifacts such as high amplitude noise. Notice that, similar to the mean, the other pooled values presented in the histograms are close, also implying small differences between the algorithms' output.

Next, we consider the full HDR-ꟻLIP error maps of the test images from algorithms A and B, shown in Figure 19-6, together with the corresponding exposure maps. In the error maps, we see that the two algorithms generate very different results, as was implied by the LDR images shown in Figure 19-4. Algorithm A generates more error in the middle third of the image, while algorithm B generally has more large errors in the top and bottom thirds of the image. The area highlighted in blue in the LDR images presented in Figure 19-4 is an example showing that B tends to blur reflected emitters. On the other hand, in the green highlighted areas in the same figure, we see that A shows particular difficulties with noise in the dark regions. The figure also indicates that A struggles more with noise than B does, in general. These issues are represented well in the HDR-ꟻLIP maps, shown in Figure 19-6. In those error maps, we also see large errors in the region corresponding to the white marked area in Figure 19-4, despite the errors being seemingly small in that figure. This is a consequence of the error becoming larger with longer exposures. We can see that this is the case by examining the exposure maps, which contain many bright colors, indicating that the maximum error is often seen at longer exposures. The medium-bright (greener) parts of the exposure

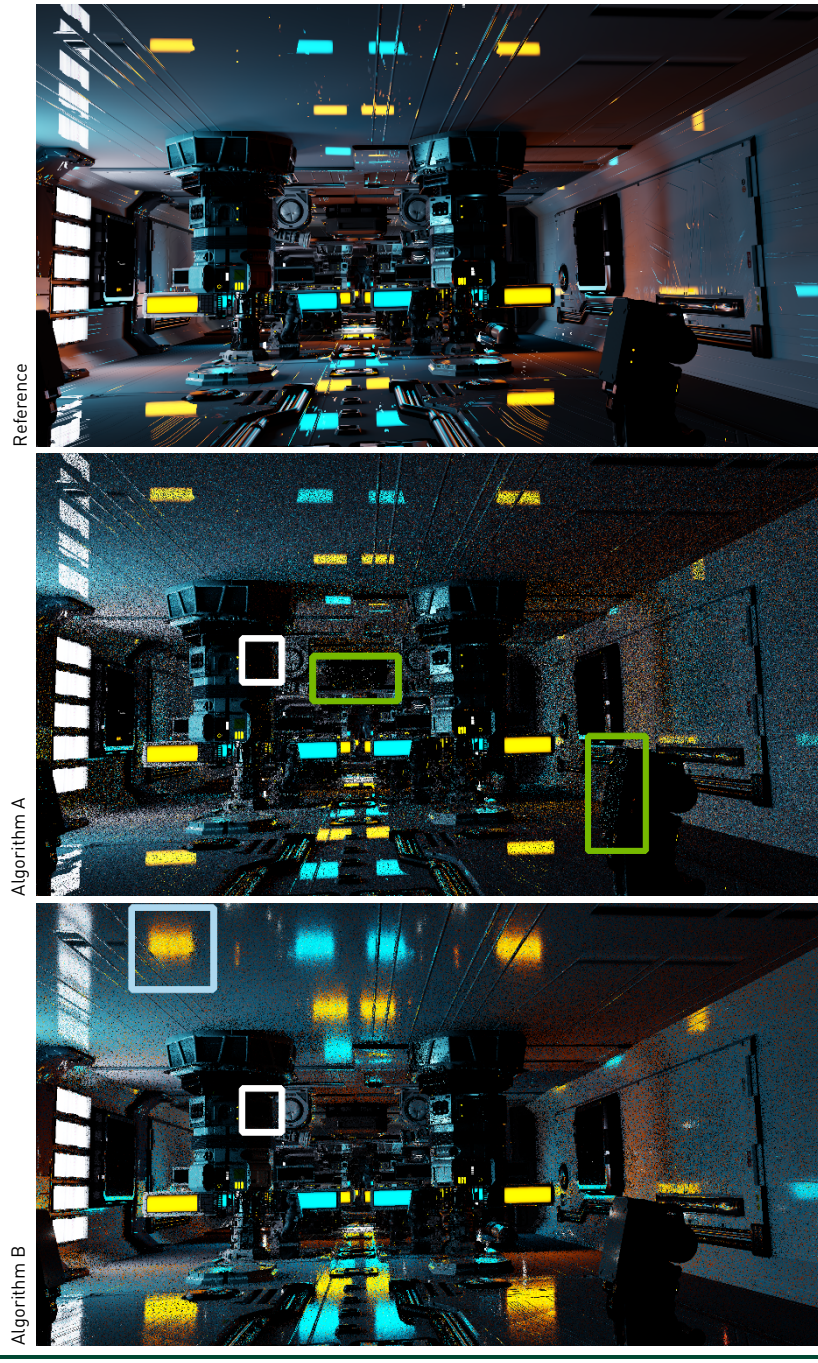


Figure 19-4. *The reference and test images for algorithms A and B, where we have marked, in green for A and in blue for B, difficult areas for the two. Within the shared, white rectangle, both algorithms seem to perform well. All images have been exposure compensated with c = 0.2 and tone mapped for display.*

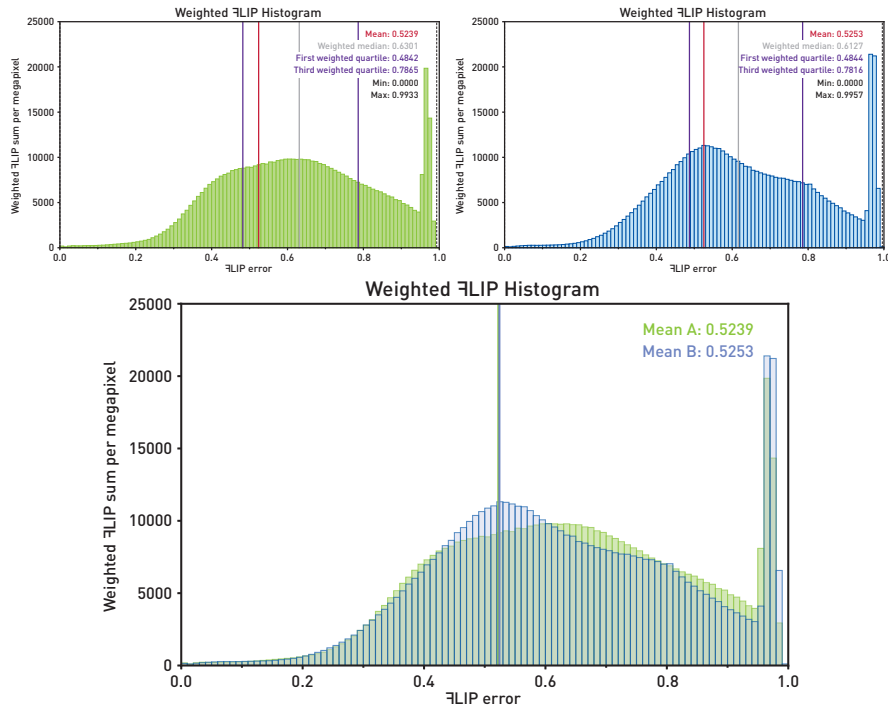


Figure 19-5. *Top: the weighted histograms of the HDR-FLIP error maps for algorithms A (left) and B (right). Bottom: the top two histograms overlapped.*

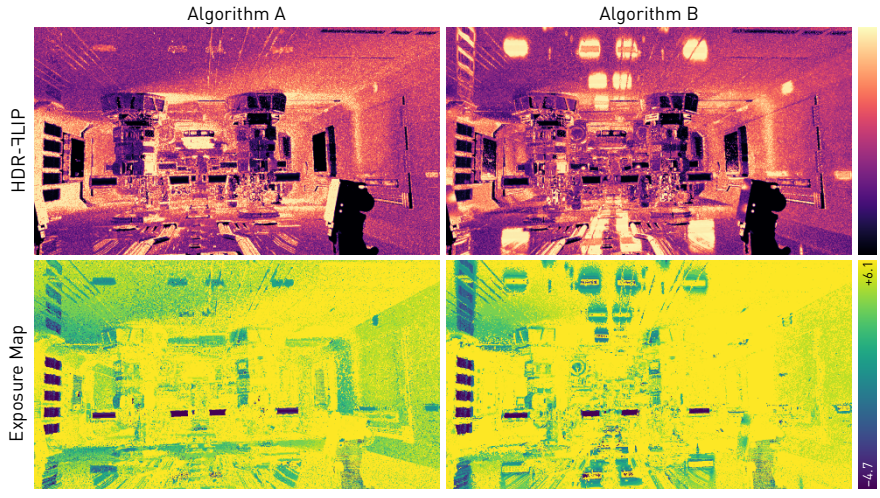


Figure 19-6. *HDR-FLIP error maps and exposure maps for algorithms A and B. The automatically computed start and stop exposures are shown in the lower right color map.*

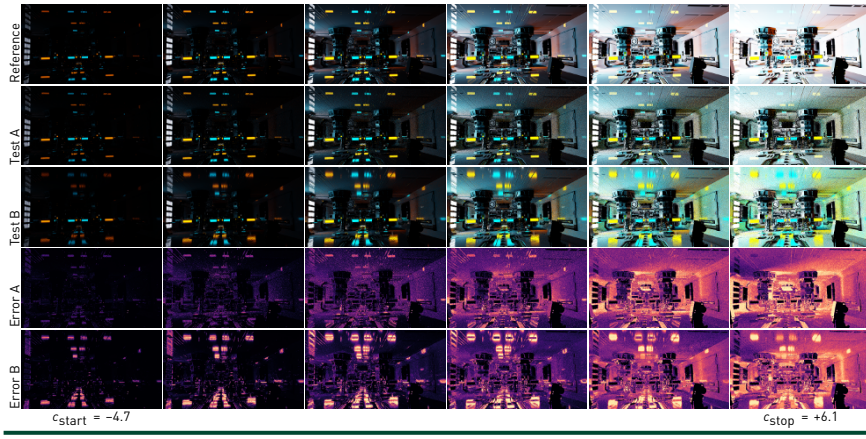


Figure 19-7. *Six out of the eleven LDR images and LDR-ꟻLIP error maps used to compute the HDR-ꟻLIP results presented in Figure 19-6. We can see how errors in some areas are larger for lower and medium exposures, whereas other areas show the largest error for higher exposures, as revealed by the exposure maps in Figure 19-6.*

maps reveal that many of the largest errors would be seen at medium-range exposures, such as the one used in Figure 19-4, whereas the dark parts of the maps indicate that short exposures are needed, too.

For completeness, we present several of the LDR images used to compute the HDR-ꟻLIP images in Figure 19-7. With these, we can confirm that the analysis we just conducted, based solely on a small set of LDR images and the HDR-ꟻLIP information, was correct and that analyzing the LDR images one by one would have led to similar conclusions. This implies that HDR-ꟻLIP yields results that allow us to correctly carry out an analysis of the error between the HDR test and reference image. We note how the pooled values, while sometimes sufficient, can be deceiving as they can be similar for two images even though the images are vastly different. They might indicate larger errors for an image that are perceived as less erroneous than another. The histograms reveal more about the error distributions and can, for example, be used to differentiate between an image with many small errors and an image with few large ones, which the mean error could not. However, they do not provide any information about where the errors are located. For this information, the full error maps are required, as was seen in the example. Thus, unless pooling is strictly necessary, it is recommended that full error maps be used when communicating the errors in the test images.

19.5 RENDERING ALGORITHM DEVELOPMENT AND EVALUATION

After analyzing the different outputs of ꟻLIP in Section 19.4, we will now look at using the results for algorithm development. In particular, we will investigate the relationship between image quality and frame time for different parameter configurations of the ReSTIR algorithm [5]. We do this by evaluating the GPU frame time and HDR-ꟻLIP errors of the different configurations and plot the results in a diagram, with frame time on one axis of the diagram and HDR-ꟻLIP mean values on the other. In this diagram, we may find the Pareto frontier. For configurations on the Pareto frontier, it holds that one dimension (e.g., image error) cannot be lowered without the second dimension (e.g., frame time) becoming worse, so this is an excellent way to provide information about the best performing configurations.

The ReSTIR algorithm features a plethora of configuration options and large parameter spans. We will focus on subsets of both. In particular, we look at the four parameters listed next while the remaining configuration is kept fixed. For details about the parameters, consult the original ReSTIR paper [5]. We run the biased version of ReSTIR as part of an optimized path tracer and render one sample per pixel (spp). In our resulting diagram, we encode the parameters so that each configuration is uniquely visualized by a marker. We refer the reader to Figure 19-8 and its legend for an example. The encoding is outlined in this list:

- *Use final visibility* indicates that a ray is traced during shading to the light that was picked. This is encoded by the shape of the markers in the diagram.

- *Use denoiser* states that a denoiser is applied to the output of ReSTIR and is encoded by the color of the markers' outlines. Unlike the original ReSTIR paper [5], we do not use the OptiX denoiser [15], but instead the ReLAX denoiser from the NVIDIA Real-Time Denoisers SDK [9].

- *#initial light samples* specifies how many initial light samples ReSTIR uses. The light sample count ranges from 1 to 32 and is recorded inside the markers.

- *#spatial neighbors* represents the number of taps that are done in one iteration of spatial reuse and is a number ranging from one to five. This is encoded by the color of the markers.

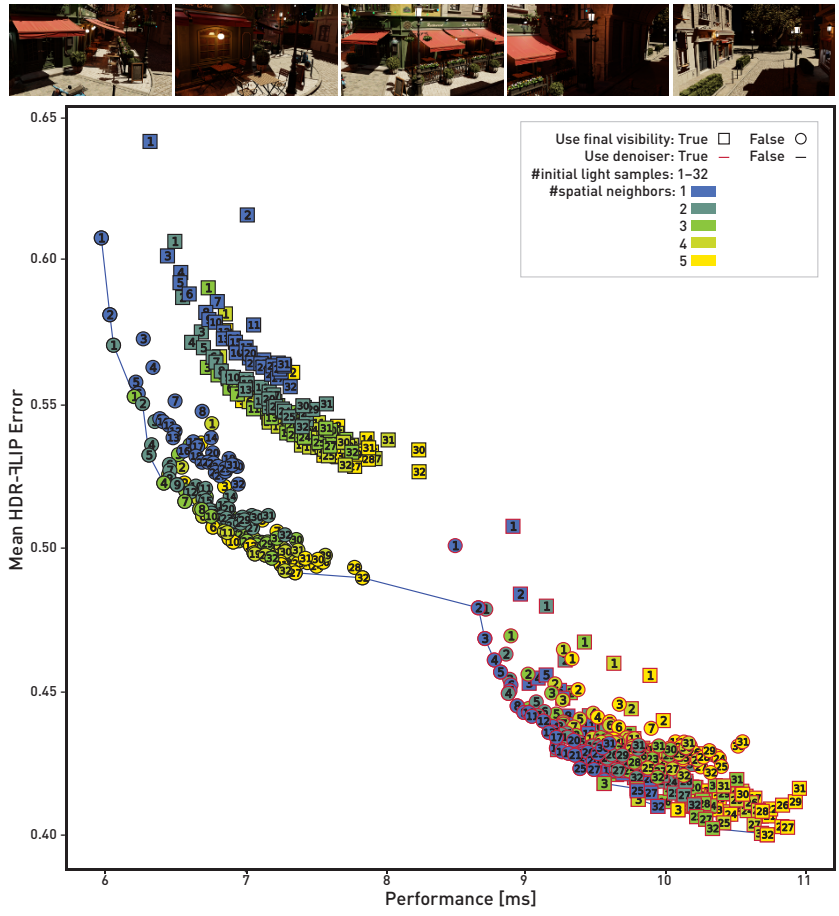


Figure 19-8. *Top: reference images of the five viewpoints used to evaluate the image quality of the ReSTIR configurations in the BistroNight scene. The images are exposure compensated with factor c = –3.0. Bottom: diagram showing the frame time and HDR-FLIP error for the different ReSTIR configurations. The properties of the markers indicate the configuration, as explained by the legend. The blue line is the Pareto frontier, which is a list of the best configurations.*

Our experiment is set up in the BistroNight scene. The quality measurements are done for frames generated using five different camera locations and angles, or *viewpoints*, namely those shown at the top in Figure 19-8. For a viewpoint, we first render a reference with two million spp, which are enough samples for the renderer to converge in the scene. Then, using a given ReSTIR configuration, we generate a test image of the same viewpoint and compute the mean HDR-FLIP error between the reference and the test image. After repeating this for each viewpoint, the final quality value of the

configuration is the average of the HDR-ꟻLIP means over the five frames. For the performance measurements, we first render frames for three minutes as a warm up period for the GPU, which in our experiments is an NVIDIA Quadro RTX A6000. Then, for each configuration, we render warm-up frames for an additional 30 seconds, followed by the recording of frame times for 30 seconds. This procedure makes for reasonably reliable rendering time evaluations. During the performance measurements, the camera moves through the scene, following a path chosen such that each of the five viewpoints used for the quality evaluation are included. The results of the experiment in the BistroNight scene are shown in the diagram in Figure 19-8.

Next, we analyze the results in Figure 19-8 by identifying different clusters of configurations. Initially, we note three large clusters of configurations, one that includes denoising (indicated by red outlines) and two that do not (black outlines). The two that do not are separated by whether or not they use final visibility (squares or circles). From the well-separated cluster that uses denoising, we see that the denoiser has a large effect on both quality and performance. Using the denoiser improves image quality but reduces performance, which is expected. Now, looking at the two clusters for which a denoiser is not used, we see that when final visibility is enabled, i.e., when a ray is traced to the light that was picked, the quality and performance are worse than when it is not enabled. However, for the configurations that use a denoiser, this is not always the case, as indicated by the squares in the bottom right in the diagram. Those squares indicate that when we use a denoiser, together with many spatial neighbors (indicated by brighter colors inside the marker) and initial light samples (number inside the marker), using final visibility leads to better quality compared to not using it. Lastly, by observing the numbers and colors in the markers, we see that, in general, an increased number of initial light samples and spatial neighbors both increase quality, but lead to longer frame times.

The blue curve in Figure 19-8 indicates the Pareto frontier. Following the frontier from the top left, where we have the shortest frame times, we see that if we do not use a denoiser, the best configurations are those that do not use final visibility (circles with black outlines). One can then optimize for quality or frame time by changing the number of initial light samples and spatial neighbors—more will reduce the error but increase frame time. Adding a denoiser may increase quality further. For low light sample counts and few spatial neighbors, the frontier indicates that final visibility should not

be used. At the end of the curve, where the image error is lowest, the configurations do, however, use final visibility, together with high light sample counts and a large number of spatial neighbors.

The results of our experiment indicate that using final visibility without a denoiser is the only choice that consistently increases both frame time and image error simultaneously. The following example, which is centered around Figure 19-9, aims to explain why this was the case. The figure shows four exposure compensated and tone mapped test images. In particular, they are insets from the fifth viewpoint used in the experiment (see Figure 19-8) and exposure compensated with the longest exposure used to compute the HDR-ꟻLIP errors. Each test image was generated with 32 initial light samples and five spatial neighbors. What differs between the configurations is whether or not they use final visibility and the denoiser. The test images are presented together with the reference and their corresponding LDR- and HDR-ꟻLIP maps. First, consider the two images that are generated without the denoiser. Comparing those, we see that the shadows are best localized in the image generated with final visibility. They are, however, much darker than those in the other image, which results in large ꟻLIP errors. The dark shadows are a consequence of pixel saturation not being possible when a pixel has the value zero. When such pixels are exposure compensated with long exposures, they remain zero, whereas pixel values that are positive increase significantly and eventually become saturated. This implies that, when we compare those pixels, the error increases with exposure, until it is maximal. On the other hand, when we compare two nonzero pixels, the error has a maximum for some exposure, after which it generally decreases after either the reference or test pixel has become saturated. We see that there are significantly fewer pixels with value zero in the test image that does not use final visibility, which results in a lower HDR-ꟻLIP mean. When we instead consider the images that are generated with a denoiser, we see that the tables are turned and using final visibility results in the highest-quality image. This follows from the fact that the denoiser spreads information between pixels and leaves few with value zero, which ultimately makes the test image generated with final visibility more similar to the reference at high exposures.

We conclude by noting that the previous experiment has shown how a developer may use the Pareto frontier to select one of many algorithms and/or configurations to achieve desired quality and performance. Lastly, we wish to repeat that developers should be cautious about drawing conclusions

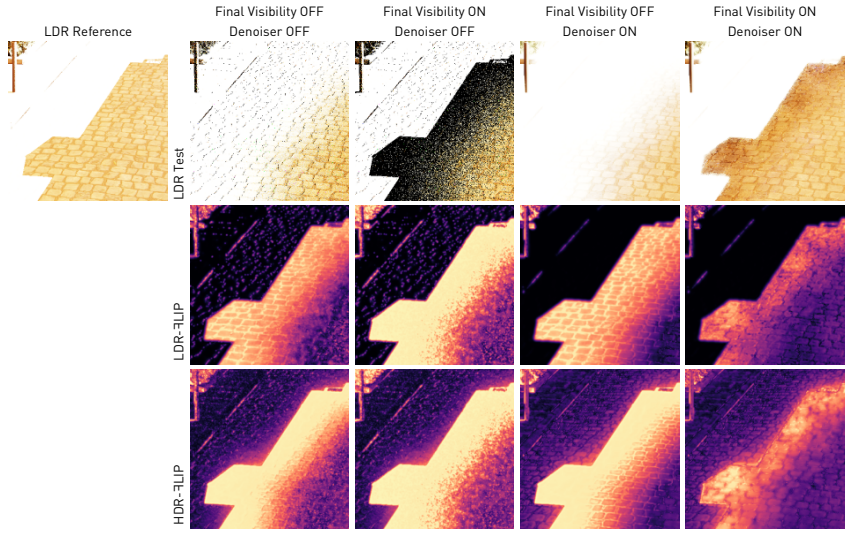


Figure 19-9. *Images including one part of the rightmost viewpoint in Figure 19-8, tone mapped under a long exposure (c = 4.7). First, we show the reference. Then, we have one column per ReSTIR configuration for four different configurations, where the columns' titles indicate the configuration. In the first row, we show the generated test image exposure compensated and tone mapped similar to the reference. The second and third rows show the LDR- and HDR-FLIP maps, respectively. We consider configurations that either use or do not use final visibility and a denoiser, while the remaining parameters are identical (32 initial light samples and five spatial neighbors). The mean HDR-FLIP errors for the full images generated with the four configurations were, from left to right,* 0.47, 0.52, 0.43, *and* 0.41. *We see that, while the shadows in the second configuration lead to large errors, adding the denoiser to it results in the image that most closely resembles the reference out of the four.*

regarding image quality based only on pooled values of error metrics, such as the mean of an LDR- or HDR-FLIP error map. Though they may give a general idea of the error between a test and corresponding reference image, only full error maps have the potential to fully describe the error.

19.6 APPENDIX: MEAN VERSUS WEIGHTED MEDIAN

In the original paper about LDR-FLIP [3], it was recommended that the weighted median be chosen if only a single error value was to be reported. There are some issues with the weighted median, however, that make its output nonintuitive. For example, if all pixels have a FLIP error of 0.0, then all pixels will fall in the first bucket. Assuming that we use the default 100 buckets, the first bucket's center point is 0.005, and thus, the weighted median will be 0.005 because we search for it in the weighted histogram. So,

even though all errors are 0.0, the reported number is not 0.0, which is counterintuitive. Alternatively, an equilibrium (center of mass) equation could be used to compute the weighted median, but then pixels with weight 0.0 would have no effect on the computation. This means that if all pixels had 0.0 error, except one that had an error of 1.0, the weighted median would be 1.0, which is also counterintuitive. Instead, we recommend using the *mean* error if a single error value is to be reported, as it returns the expected results in the cases discussed and is a well-known and easily computed measure.

ACKNOWLEDGMENTS

We are grateful to the following who let us use their assets: ZeroDay ©beeple [19], Attic (https://developer.nvidia.com/usd), and BistroNight ©Amazon Lumberyard [2]. Thanks to Peter Shirley for cheering us on to write this chapter and to Chris Wyman and Pawel Kozlowski for helping us navigate the parameter space of ReSTIR.

REFERENCES

[1] Abasi, S., Tehran, M. A., and Fairchild, M. D. Distance metrics for very large color differences. *Color Research and Application*, 45(2):208–223, 2020. DOI: 10.1002/col.22451.

[2] Amazon Lumberyard. Amazon Lumberyard Bistro. *Open Research Content Archive (ORCA)*, http://developer.nvidia.com/orca/amazon-lumberyard-bistro, 2017.

[3] Andersson, P., Nilsson, J., Akenine-Möller, T., Oskarsson, M., Åström, K., and Fairchild, M. D. ꟻLIP: A difference evaluator for alternating images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2):15:1–15:23, 2020. DOI: 10.1145/3406183.

[4] Andersson, P., Nilsson, J., Shirley, P., and Akenine-Möller, T. Visualizing errors in rendered high dynamic range images. In *Eurographics Short Papers*, pages 25–28, 2021. DOI: 10.2312/egs.20211015.

[5] Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A., and Jarosz, W. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics*, 39(4):148:1–148:17, July 2020. DOI: 10.1145/3386569.3392481.

[6] Čadík, M., Herzog, R., Mantiuk, R., Myszkowski, K., and Seidel, H.-P. New measurements reveal weaknesses of image quality metrics in evaluating graphics artifacts. *ACM Transactions of Graphics*, 31(6):147:1–147:10, 2012. DOI: 10.1145/2366145.2366166.

[7] CIE Comission Internationale de l'Eclairage. Recommendations on uniform color spaces, color-difference equations, psychometric color terms. Supplement No. 2 to CIE publication No. 15 (E.-1.3.1) 1971/(TC-1.3.), 1978.

[8] Hunt, R. W. G. Light and dark adaptation and the perception of color. *Journal of the Optical Society of America*, 42(3):190–199, 1952.

[9] Kozlowski, P. and Cheblokov, T. ReLAX: A denoiser tailored to work with the ReSTIR algorithm. GPU Technology Conference, https://developer.nvidia.com/nvidia-rt-denoiser, 2021.

[10] Liu, Y. and Heer, J. Somewhere over the rainbow: An empirical assessment of quantitative colormaps. In *CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2018.

[11] Mantiuk, R., Kim, K. J., Rempel, A. G., and Heidrich, W. HDR-VDP-2: A calibrated visual metric for visibility and quality predictions in all luminance conditions. *ACM Transactions on Graphics*, 30(4):40:1–40:14, 2011. DOI: 10.1145/1964921.1964935.

[12] Munkberg, J., Clarberg, P., Hasselgren, J., and Akenine-Möller, T. High dynamic range texture compression for graphics hardware. *ACM Transactions on Graphics*, 25(3):698–706, 2006. DOI: 10.1145/1141911.1141944.

[13] Narkowicz, K. ACES filmic tone mapping curve. https://knarkowicz.wordpress.com/2016/01/06/aces-filmic-tone-mapping-curve/, January 6, 2016.

[14] Nilsson, J. and Akenine-Möller, T. Understanding SSIM. arXiv:eess.IV, https://arxiv.org/abs/2006.13846, 2020.

[15] NVIDIA Research. NVIDIA Optix™ AI-accelerated denoiser. https://developer.nvidia.com/optix-denoiser, 2017.

[16] Rousselle, F., Knaus, C., and Zwicker, M. Adaptive sampling and reconstruction using greedy error minimization. *ACM Transactions on Graphics*, 30(6):159:1–159:12, 2011. DOI: 10.1145/2024156.2024193.

[17] Vogels, T., Rousselle, F., Mcwilliams, B., Röthlin, G., Harvill, A., Adler, D., Meyer, M., and Novák, J. Denoising with kernel prediction and asymmetric loss functions. *ACM Transactions of Graphics*, 37(4):124:1–124:15, 2018. DOI: 10.1145/3197517.3201388.

[18] Wang, Z., Bovik, A. C., Sheikh, H. R., and Simoncelli, E. P. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004. DOI: 10.1109/TIP.2003.819861.

[19] Winkelmann, M. Zero-Day. *Open Research Content Archive (ORCA)*, https://developer.nvidia.com/orca/beeple-zero-day, 2019.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.


XU QIONG
LONDON
HAIR SALON
ELEK
DRIVER
EV22 LQQ



PART III
SAMPLING

PART III

SAMPLING

Ray tracing is almost synonymous with sampling. After all, tracing a ray is the same as sampling a point on a piece of geometry. But the number of rays we can trace within each frame is limited, so the money is on the question: to where should we trace these rays?

The chapters in this part all provide different, complementary answers, ranging from simple techniques that are useful in any practitioner's toolbox to advanced algorithms that efficiently render millions of light sources.

Chapter 20, *Multiple Importance Sampling 101*, introduces a very powerful, yet simple, procedure for combining multiple sampling strategies such that the strengths of each are preserved. Multiple importance sampling is a fundamental tool that can be utilized to great effect in almost any situation where samples have to be drawn. This chapter gently introduces the topic by the example of combining material and light sampling in direct illumination, culminating in a fully fledged low-noise path tracing algorithm.

Chapter 21, *The Alias Method for Sampling Discrete Distributions*, presents a constant-time algorithm for randomly selecting an element with predetermined probability from a potentially large set. It is one of those algorithms that once you know about it, you keep finding a use case for it: be that sampling a texel proportional to its brightness or randomly selecting a light source proportional to its power.

However, sometimes the set of possible samples is not predetermined, and instead each sample arrives one by one as a stream of data. For this situation, Chapter 22, *Weighted Reservoir Sampling: Randomly Sampling Streams*, describes an algorithm for sampling elements from such a stream in a single pass. This technique was only recently introduced to ray tracing, but it has already shown great promise by allowing the efficient reuse of "good" samples from neighboring pixels or even previous frames.

Combined, the techniques just listed can make up sophisticated rendering algorithms that can handle seemingly impossible situations. Chapter 23, *Rendering Many Lights with Grid-Based Reservoirs*, presents a holistic

approach to tackle the difficult problem of selecting only the few most important out of millions of light sources, which enables visual effects that were infeasible to get in real time prior to ray tracing.

Lastly, it is important to consider not only the probabilities of samples but also the pattern in which they are arranged. Chapter 24, *Using Blue Noise for Ray Traced Soft Shadows*, is a deep dive into the world of sample patterns with blue-noise characteristics, showing how such samples may lead not only to lower rendering error compared to independent samples, but also to images that are easier to denoise and that look more visually pleasing.

After reading this part, you will have learned a few versatile tools for deciding where to trace these oh-so-valuable rays. But the adventure is far from over, because the elusive theoretically optimal sampling procedure—the one that has zero noise—will always keep beckoning for a newer, better algorithm.

Thomas Müller

CHAPTER 20

MULTIPLE IMPORTANCE SAMPLING 101

Anders Lindqvist
NVIDIA

ABSTRACT

We present the basics of *multiple importance sampling* (MIS), a well-known algorithm that can lower the amount of noise in a path tracer significantly by combining multiple sampling strategies. We introduce MIS in the context of direct light sampling and show how it works in a path tracer. We assume that the reader has already written a simple path tracer and wants to improve it.

20.1 DIRECT LIGHT ESTIMATION

We will be focusing on the problem of direct lighting in a path tracer. The content is heavily inspired by Veach's doctoral thesis [4]. Our problem statement is that for a given surface point $\mathbf{P}$ with a normal $\mathbf{n}$ and a material, we want to estimate the direct light being reflected in a direction $\omega_\mathbf{o}$. We will use code to illustrate all concepts, but the direct light integral we are estimating is

$$L_r(\mathbf{P}, \omega_\mathbf{o}) = \int_\Omega \underbrace{L_e(r(\mathbf{P}, \omega_\mathbf{i}), -\omega_\mathbf{i})}_{\text{light}} \underbrace{f(\mathbf{P}, \omega_\mathbf{o}, \omega_\mathbf{i})(\mathbf{n} \cdot \omega_\mathbf{i})}_{\text{BRDF and cosine}} d\omega_\mathbf{i}, \tag{20.1}$$

where r is a ray tracing function that shoots a ray into the scene and picks the first visible surface (if any), L_e is the emission color at a surface point, and f is the *bidirectional reflectance distribution* (BRDF) that describes how light is reflected at the surface.[1] The integral is over all directions $\omega_\mathbf{i}$ from which lights come over the hemisphere Ω around the normal. The word *estimate* here is used in a Monte Carlo sense in that we want to be able to average several such estimates to get a better estimate.

A first path tracer will most likely employ one of two techniques to handle direct lighting: material sampling or light sampling. We will describe them

[1] Here, we assume that if r does not hit a surface, it causes L_e to become black.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_20

Listing 20-1. *Direct light using cosine hemisphere sampling.*

```
vec3 direct_cos(vec3 P, vec3 n, vec3 wo, Material m) {
  vec3 wi = random_cosine_hemisphere(n);
  float pdf = dot(wi, n)/PI;
  Intersect i = intersect(P, wi);
  if (!i.hit) return vec3(0.0);
  vec3 brdf = evaluate_material(m, n, wo, wi);
  vec3 Le = evaluate_emissive(i, wi);
  return brdf*dot(wi, n)*Le/pdf;
}
```

both and discuss their pros and cons. Though we only talk about emissive surfaces in our examples, the same methods can be used with a sky dome or skylight as well.

20.1.1 COSINE HEMISPHERE SAMPLING

We start with an even more basic sampling variant. In cosine hemisphere sampling we use no knowledge of the material or the light sources when choosing samples. Instead, we only sample according to the $\mathbf{n} \cdot \omega_\mathbf{i}$ term in Equation 20.1. Choosing directions in a way that is not uniform over the hemisphere is called *importance sampling* and is directly supported by the Monte Carlo integration framework.

The code in Listing 20-1 shoots one ray into the scene and gives out an estimate. Because this is the first Monte Carlo integrator, we show a full evaluation using `N` samples for a given surface point `P`, normal `n`, and material `mat`.

```
vec3 estimate = vec3(0.0);
for (int i=0; i<N; i++) {
  estimate += direct_cos(P, n, mat);
}
estimate /= N;
```

When we go to full path tracing, we will use a different surface point for each sample in order to get antialiasing, but the principles are the same when estimating the integral. Note that the two dot products in Listing 20-1 cancel out and, as we only generate directions in the hemisphere around the normal, there is no need to clamp the dot products.

20.1.2 MATERIAL SAMPLING

In material sampling we select directions with a probability that is proportional to $f(\mathbf{P}, \omega_\mathbf{o}, \omega_\mathbf{i})(\mathbf{n} \cdot \omega_\mathbf{i})$, as shown in Listing 20-2. For a fully diffuse

Listing 20-2. *Direct light using material sampling.*

```
vec3 direct_mat(vec3 P, vec3 n, vec3 wo, Material m) {
  vec3 wi;
  float pdf;
  if (!sample_material(m, n, wo, &wi, &pdf)) {
    return vec3(0.0);
  }
  Intersect i = intersect(P, wi);
  if (!i.hit) return vec3(0.0);
  vec3 brdf = evaluate_material(m, n, wo, wi);
  vec3 Le = evaluate_emissive(i, wi);
  return brdf*dot(wi, n)*Le/pdf;
}
```

Listing 20-3. *Direct light using light sampling.*

```
float geom_fact_sa(vec3 P, vec3 P_surf, vec3 n_surf) {
  vec3 dir = normalize(P_surf - P);
  float dist2 = distance_squared(P, P_surf);
  return abs(-dot(n_surf, dir)) / dist2;
}

vec3 direct_light(vec3 P, vec3 n, vec3 wo, Material m) {
  float pdf;
  vec3 l_pos, l_nor, Le;
  if (!sample_lights(P, n, &l_pos, &l_nor, &Le, &pdf)) {
    return vec3(0.0);
  }
  float G = geom_fact_sa(P, l_pos, l_nor);
  vec3 wi = normalize(l_pos - P);
  vec3 brdf = evaluate_material(m, n, wo, wi);
  return brdf*G*clamped_dot(n, wi)*Le/pdf;
}
```

material, this is exactly what we do with the cosine hemisphere sampling technique. For a more glossy material, this could mean that we more often select directions close to the reflection direction.

20.1.3 LIGHT SAMPLING

In light sampling we pick positions on the light sources themselves. This is different from cosine hemisphere sampling or material sampling where we do not use any scene information (except the surface normal and surface material in material sampling).

In Listing 20-3 the sampling of light sources chooses points on surfaces in the scene. In order to be compatible with Equation 20.1, we must apply a

geometric factor G [4, Equation 8.3], which may look familiar to anyone who has implemented light sources in real-time graphics. It accounts for the fact that if we choose a point on a surface, we must take attenuation into account; points far away should be darker. If we look at sampling using directions—like we do for cosine hemisphere and material sampling—there is nothing in there. Instead, it is implicit in the fact that we hit things far away with a lower probability. In Listing 20-3 we see the factor `G` at the end. If there is something blocking the emissive surface (like another surface, emissive or not), `sample_light` will return false and we will not get any contribution. Note that unlike Veach we have chosen to not include `dot(n, wi)` in G. This is reflected in our function name `geom_fact_sa`.

A first implementation of `sample_lights` in a scene where all the light sources are spheres or quads could look something like Listing 20-4.

Listing 20-4. *Sample light sources.*

```
bool sample_lights(vec3 P, vec3 n, vec3 *P_out, vec3 *n_out,
  vec3 *Le_out, float *pdf_out)
{
  int chosen_light = floor(uniform()*NUM_LIGHTS);
  vec3 l_pos, l_nor;
  float p = 1.0 / NUM_LIGHTS;
  Object l = objects[chosen_light];

  if (l.type == GeometryType::Sphere) {
    float r = l.size.x;
    // Choose a normal on the side of the sphere visible to P.
    l_nor = random_hemisphere(P-l.pos);
    l_pos = l.pos + l_nor * r;
    float area_half_sphere = 2.0*PI*r*r;
    p /= area_half_sphere;
  } else if (l.type == GeometryType::Quad) {
    l_pos = l.pos + random_quad(l.normal, l.size);
    l_nor = l.normal;
    float area = l.size.x*l.size.y;
    p /= area;
  }

  bool vis = dot(P-l_pos, l_nor) > 0.0; // Light front side
  vis &= dot(P-l_pos, n) < 0.0; // Behind the surface at P
  // Shadow ray
  vis &= intersect_visibility(safe(P, n), safe(l_pos, l_nor));

  *P_out = l_pos;
  *n_out = l_nor;
  *pdf_out = p;
  *Le_out = vis ? l.material.emissive : vec3(0.0);
  return vis;
}
```

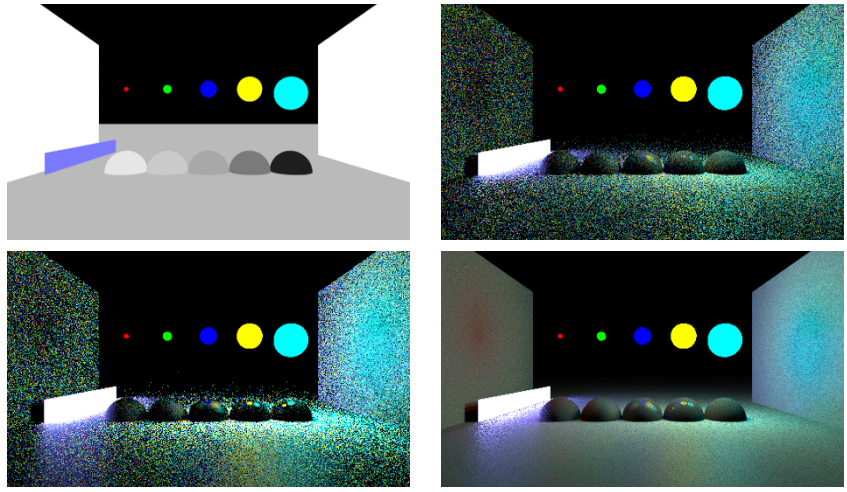


Figure 20-1. *Scene description (upper left), cosine hemisphere sampling (upper right), material sampling (lower left), and light sampling (lower right). Roughness is shown in the first image with black being low roughness and white being high roughness. The color of the emissive light sources are shown as well. We can see that material sampling best captures the sharp reflections of the emissive spheres in the rightmost half-sphere. Light sampling has far less noise but struggles near the emissive quad on the left. All images are rendered using the same amount of samples.*

The function `safe` offsets the position by the normal such that it is moved out of self-intersection.

20.1.4 CHOOSING A TECHNIQUE

Light sampling works very well for scenes with diffuse surfaces but less so for scenes with glossy materials. If the BRDF only includes incoming light in a small region on the hemisphere, it is unlikely that light sampling will pick directions in that region. Light sampling also struggles where emissive surfaces are very near the position from which we sample. This can be seen at the base of the emissive quad in Figure 20-1 where there is noise. Material sampling works best when we have highly reflective surfaces and large light sources that are easy to hit. In Figure 20-1 we can see that only material sampling finds the emissive surfaces on the reflective half-spheres with low roughness.

When we fail to pick the best strategy, we are punished with noise. This happens because we find the light sources with a direction that is relevant for the BRDF only with a very low probability. Once we get there, we *boost* the contribution a lot by dividing by the really low probability. We could try to pick

a technique based on the roughness value of the surface, but it is very error prone because the optimal choice also depends on the size and proximity of the emissive surfaces. Per-scene tweaks of thresholds are never fun, and here we might even have to tweak on an even finer level.

20.1.5 MULTIPLE IMPORTANCE SAMPLING

Let us start by considering what would happen if we would render the image once using each technique and blend them together evenly. What would happen for pixels where one technique was good but the other ones were really bad? Unfortunately, there is nothing from the noise-free images that "masks" the noise from the bad images. The noise would still be there, only slightly dimmed by blending in the good image. Like we discussed in the previous section, there is no optimal strategy even for a single surface point; the choice will be different on different parts of the hemisphere!

A better way was shown in Chapter 9 of Veach's thesis [4]. It shows how multiple importance sampling can be used to combine multiple techniques. The idea is that whenever we want to estimate the direct light, we use *all* techniques! We will assume that all of our techniques generate a surface point and give us a probability of choosing that surface point. With MIS we let each technique generate a surface point. Then we compare the probability of generating that surface point with the probability that the other techniques would generate that same point. If the other techniques can generate that surface point with a higher probability, we let them handle most of it, maybe all of it.

To determine the per-sample-per-technique weights, we need some sort of heuristic. Veach introduced a heuristic called the *balance heuristic* and proved that is close to optimal. See Listing 20-5.

Listing 20-5. *Balance heuristics for two techniques.*

```
1 float balance_heuristic(float pdf, float pdf_other) {
2   return pdf / (pdf + pdf_other);
3 }
```

By plugging in the probability from the material sampling technique and the light sampling technique into the balance heuristic, we can determine the weights to use. Almost. First, we need to adapt the material sampling to also generate surface points. Material sampling generates a direction. By shooting the ray into the scene, we can find the surface point and normal

representing that sampled direction. The probability of choosing that surface point then can be obtained by combining the probability of choosing the direction and the geometric factor *G* like we did for light sampling earlier, as shown in Listing 20-6.

Listing 20-6. *Direct light using MIS.*

```
vec3 direct_mis(vec3 P, vec3 n, vec3 wo, Material m) {
  vec3 result = vec3(0.0);
  vec3 Le, m_wi, l_pos, l_nor;
  float l_pdf, m_pdf;
  // Light sampling
  if (sample_lights(P, n, &l_pos, &l_nor, &Le, &l_pdf)) {
    vec3 l_wi = normalize(l_pos - P);
    float G=geom_fact_sa(P, l_pos, l_nor);
    float m_pdf=sample_material_pdf(m, n, wo, l_wi);
    float mis_weight=balance_heuristic(l_pdf, m_pdf*G);
    vec3 brdf=evaluate_material(m, n, wo, l_wi);
    result+=brdf*mis_weight*G*clamped_dot(n, l_wi)*Le/l_pdf;
  }
  // Material sampling
  if (sample_material(m, n, wo, &m_wi, &m_pdf)) {
    Intersect i = intersect(P, m_wi);
    if (i.hit && i.mat.is_emissive) {
      float G=geom_fact_sa(P, i.pos, i.nor);
      float light_pdf=sample_lights_pdf(P, n, i);
      float mis_weight=balance_heuristic(m_pdf*G, light_pdf);
      vec3 brdf=evaluate_material(m, n, wo, m_wi);
      vec3 Le=evaluate_emissive(i, m_wi);
      result+=brdf*dot(m_wi, n)*mis_weight*Le/m_pdf;
    }
  }
  return result;
}
```

The function `sample_lights_pdf` in Listing 20-7 says at what probability the light sampling would choose this emissive surface point, given that it was queried with the position and normal at which we are currently evaluating.

Listing 20-7. *Sample lights pdf.*

```
float sample_lights_pdf(vec3 P, vec3 n, Intersect emissive_surface) {
  Object object = objects[emissive_surface.object_index];
  if (!object.material.is_emissive) return 0.0;
  float p = 1.0/NUM_LIGHTS;
  if (object.type == GeometryType::Sphere) {
    float r = object.size.x;
    float area_half_sphere = 2.0*PI*r*r;
    p /= area_half_sphere;
  } else if (object.type == GeometryType::Quad) {
    float area_quad = object.size.x * object.size.y;
    p /= area_quad;
  }
  return p;
}
```

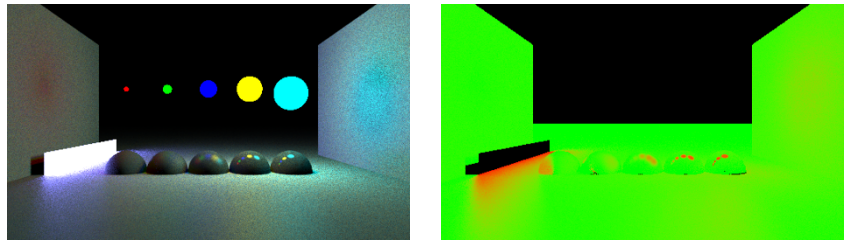


Figure 20-2. *MIS (left) and MIS weights (right). Red means that material sampling had larger weights; green means that light sampling had larger weights.*

It should give out the same probability for a surface as `sample_lights`. If `sample_lights` has an advanced scheme to select lights and adjust `p_choose_light` accordingly, then `sample_lights_pdf` must replicate that so it can give out the same probability. The same goes for `sample_materials_pdf`.

We can see the result in Figure 20-2 for our example with two techniques. There, we can also see the average MIS weight being used for the two techniques with green for light sampling and red for material sampling. If we compare Figure 20-1 with Figure 20-2, we see that we get less noise in the MIS render close to emissive objects. We also manage to find the emissive objects in the highly reflective spheres, something that light sampling failed to do. This is where material sampling is better than light sampling. Unlike material sampling we still have the low noise result that we did with light sampling when we move away from the emissive objects. It should be noted that while MIS give us a very practical way to combine multiple techniques, it would be far better if we could use a technique that choose samples according to the full product of both the material and the light sources—and maybe even taking the visibility into account!

We can think of MIS as doing two separate renders: one image using material sampling with the material MIS weight applied and one image using light sampling with the light sampling MIS weight applied. These two images are independent in the sense that they can choose completely different directions or surface points during sampling. They do not even have to use the same number of samples! The idea is just that they each do what they do best and leave the rest to the other render. Two such renders using each technique with MIS weight applied can be seen in Figure 20-3.

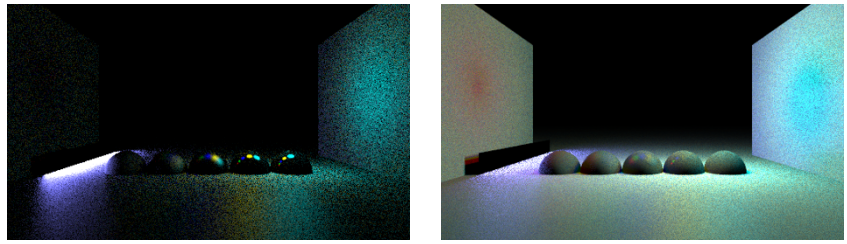


Figure 20-3. *Material sampling contribution (left) and light sampling contribution (right) (scaled by a factor of two).*

The choice to do MIS with a measure over surfaces instead of over solid angles was done to simplify this chapter. It is possible to do both, but the alternative is more complicated to describe. Our choice also makes it easy to use it in our path tracer in the next section. If we had instead chosen to do MIS using a measure for probabilities over solid angles, then we could still use the balance heuristic; it is usable as long as the probabilities use the same measure. Veach has a more general form of the balance heuristic where it can be used with any number of techniques and also with a different amount of samples from each technique. There are also alternatives to the balance heuristic [4, Section 9.2.3], but these are outside of the scope of this chapter.

20.2 A PATH TRACER WITH MIS

Now it is time to put it all together to get a full path tracer! We assume that you have written a path tracer before, so we will mostly add in the MIS of direct light in Listing 20-8.

The throughput—`tp`—says how future contributions should be weighted. If a path encounters a dark material, `tp` will be lower after bouncing the light.

The result can be seen in Figure 20-4 where we compare light sampling and MIS. Because we can reuse scene intersection from the material sampling ray to also do our bounce lighting, the only major extra work compared to a path tracer with only light sampling is the call to `sample_lights_pdf`, which is only invoked when our bounce light ends up at an emissive surface. So if you are lucky, your path tracer could get MIS more or less for free!

20.3 CLOSING WORDS AND FURTHER READING

In this chapter we discussed using MIS to handle direct lighting, but it can also be used to combine multiple ways to do bounce lighting or to balance

Listing 20-8. *Path tracing using MIS.*

```
vec3 pathtrace_mis(vec3 P, vec3 n, vec3 wo, Material m) {
  vec3 result = vec3(0), tp = vec3(1);
  while (true) {
    vec3 Le, m_wi, l_pos, l_nor;
    float l_pdf, m_pdf;
    // Light sampling
    if (sample_lights(P, n, &l_pos, &l_nor, &Le, &l_pdf)) {
      vec3 l_wi = normalize(l_pos - P);
      float G=geom_fact_sa(P, l_pos, l_nor);
      float m_pdf=sample_material_pdf(m, n, wo, l_wi);
      float mis_weight=balance_heuristic(l_pdf, m_pdf*G);
      vec3 brdf=evaluate_material(m, n, wo, l_wi);
      result+=tp*brdf*G*clamped_dot(n, l_wi)*mis_weight*Le/l_pdf;
    }
    // For material sampling and bounce
    if (!sample_material(m, n, wo, &m_wi, &m_pdf)) {
      break;
    }
    Intersect i = intersect(safe(P, n), m_wi);
    if (!i.hit) {
      break; // Missed scene
    }
    tp*=evaluate_material(m, n, wo, m_wi)*dot(m_wi, n)/m_pdf;
    if (i.mat.is_emissive) {
      float G = geom_fact_sa(P, i.pos, i.nor);
      float light_pdf=sample_lights_pdf(P, n, i);
      float mis_weight=balance_heuristic(m_pdf*G, light_pdf);
      vec3 Le = evaluate_emissive(i, m_wi);
      result+=tp*mis_weight*Le;
      break; // Our emissive surface doesn't bounce.
    }

    if (russian_roulette(&tp)) break;

    // Update state for next bounce; tp captures material and pdf.
    P = i.pos;
    n = i.nor;
    wo = -m_wi;
    m = i.mat;
  }
  return result;
}
```

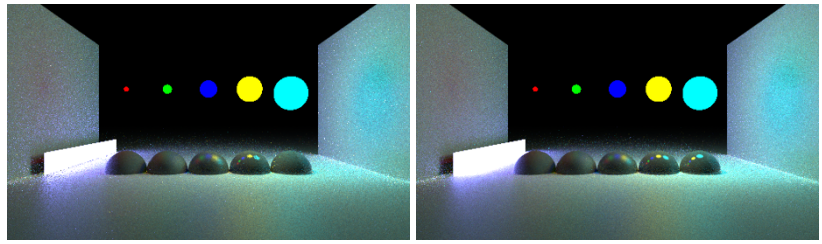


Figure 20-4. *Path tracing using light sampling (left) and MIS (right).*

different techniques to do volumetric lighting. It is a tool that can be applied whenever you have multiple techniques that each have pros and cons but that could work well together!

The concepts shown here are explained very well in Veach's doctoral thesis [4]. Kondapaneni et al. [1] proved that we can improve on the balance heuristic if we also allow negative MIS weights. Shirley et al. [3] show how to make better light sampling strategies for many types of individual light sources, and Moreau and Clarberg [2] present a system supporting scenes with many lights sources.

ACKNOWLEDGMENTS

Thanks to my editor Thomas Müller for extensive comments and mathematics help as well as source code for the reflective material model. Thanks to Alex Evans for reading and commenting on drafts, and lastly thanks to Petrik Clarberg for discussions on how to best present this topic.

REFERENCES

[1] Kondapaneni, I., Vévoda, P., Grittmann, P., Skřivan, T., Slusallek, P., and Křivánek, J. Optimal multiple importance sampling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2019)*, 38(4):37:1–37:14, July 2019. DOI: 10.1145/3306346.3323009.

[2] Moreau, P. and Clarberg, P. Importance sampling of many lights on the GPU. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 18. Apress, 2019. http://raytracinggems.com.

[3] Shirley, P., Wang, C., and Zimmerman, K. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics*, 15(1):1–36, Jan. 1996. DOI: 10.1145/226150.226151.

[4] Veach, E. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1997.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 21

THE ALIAS METHOD FOR SAMPLING DISCRETE DISTRIBUTIONS

Chris Wyman
NVIDIA

ABSTRACT

The *alias method* is a well-known algorithm for constant-time sampling from arbitrary, discrete probability distributions that relies on a simple precomputed lookup table. We found many have never learned about this method, so we briefly introduce the concept and show that such lookup tables can easily be generated.

21.1 INTRODUCTION

When rendering, we often need to sample discrete probability distributions. For example, we may want to sample from an arbitrary environment map proportional to the incoming intensity. One widely used technique inverts the cumulative distribution function [2]. This is ideal for analytically invertible functions but requires an $O(\log N)$ binary search when inverting tabulated distributions like our environment map.

If you can afford to precompute a lookup table, the alias method [5] provides a simple, constant-time algorithm for sampling from arbitrary discrete distributions. However, precomputation makes it less desirable if taking just a few samples from a distribution or if the distribution continually changes.

21.2 BASIC INTUITION

Let's start by reviewing how we often generate samples without the alias method. Imagine a discrete distribution, as in Figure 21-1. Bins have arbitrary real-valued weights, and we want to randomly sample proportional to their relative weights.

Using traditional cumulative distribution function (CDF) inversion, we first build a discrete CDF, e.g., in Figure 21-1. We pick a uniform random number

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_21

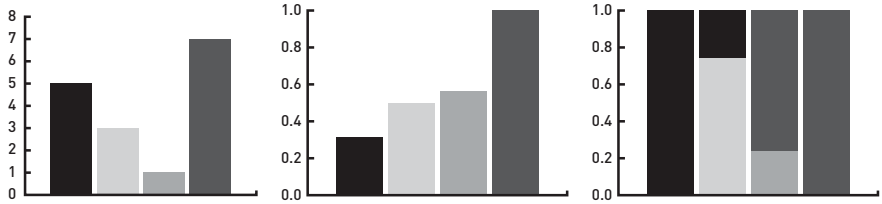


Figure 21-1. *Left: a simple discrete distribution. Center: its discrete cumulative distribution function. Right: an alias table for the distribution.*

$\xi \in [0 \ldots 1)$, place it on the *y*-axis, and select as our sample the first bin it falls into (moving left to right). Our example has weights summing to 16, so we choose the black bin if $\xi \leq \frac{5}{16}$, the light gray bin if $\frac{5}{16} < \xi \leq \frac{1}{2}$, etc. Here, finding a bin corresponding to ξ is straightforward, with at most three comparisons. But if sampling from a light probe, linearly searching each bin is costly. Even a fast binary search uses dozens of comparisons and walks memory incoherently.

Essentially, table inversion picks a random value then walks the tabulated CDF asking, "do you correspond to my sample?" Optimization involves designing a good search.

Instead, the alias method asks, "wouldn't it be easier with uniform weights?" You could easily pick a random bin without a search. Clearly, our weights are unequal. But imagine sampling bins as if uniformly weighted, say using the average weight. We would oversample those weighted below average and undersample those weighted above average.

Perhaps we could correct this error after selection? If we pick an oversampled bin, we could sometimes switch to an undersampled bin. This seems feasible; if sampling proportional to average weight, there's an equal amount of under- and oversampling.

Figure 21-1 also shows a table reorganizing weights from our distribution into equal-sized bins. We call this an *alias table* (though that term is a bit overloaded).

To sample, first choose a bin uniformly. Use random $\xi \in [0 \ldots 1)$ to check whether to switch due to oversampling, based on threshold τ. If $\xi \geq \tau$, switch to a different bin.

21.3 THE ALIAS METHOD

Now let's formalize this basic intuition. The alias method consumes N elements with positive weights ω_i. It samples from them, proportional to their relative weights, in constant time—two uniform random variates and one table lookup per sample.

A formal alias table is an array of N triplets $\{\tau, i, j\}$, where τ is a threshold in $[0 \ldots 1)$ and i and j are samples or sample indices. With care i may be stored implicitly, so that table entries are tuples $\{\tau, j\}$. Basic tables do not store or reconstruct weights ω_i.

To sample via the alias method, uniformly select $\xi_1, \xi_2 \in [0 \ldots 1)$. Use ξ_1 to pick from the N triplets, i.e., select triplet located at position $\lfloor \xi_1 N \rfloor$. Then, return i if $\xi_2 < \tau$, otherwise return j.

Interestingly, only two choices i and j are ever needed in any bin. In fact, proving this is the only tricky part of the alias method. This property arises *by construction* during table creation, though it subtly breaks the visual intuition developed in the previous section.

21.4 ALIAS TABLE CONSTRUCTION

Our goal is not efficient construction, but providing a clear and correct algorithm. Optimal $O(N)$ table construction algorithms are left for further reading. We give a simple divide-and-conquer approach similar to a proof by induction.

We start with our N input samples and their weights ω_i. We can easily compute the average sample weight $\langle\omega\rangle$; the summed weight is $\sum_{i \in N} \omega_i \equiv N\langle\omega\rangle$. Here is the idea for our simple construction: With a series of N steps, create one tuple $\{\tau, i, j\}$ per step. Each tuple has aggregate weight $\langle\omega\rangle$, so after one step we have $N - 1$ samples left to insert, with total weight $N\langle\omega\rangle - \langle\omega\rangle$ (still with an average of $\langle\omega\rangle$ per sample).

Each of these N steps looks as follows:

1. Sort the (remaining) input samples by weight.
2. Create a tuple $\{\tau, i, j\}$, where i is the lowest-weighted sample, τ is $\omega_i/\langle\omega\rangle$, and j is the highest-weighted sample.
3. Remove sample i from the list of remaining samples.
4. Reduce sample j's weight ω_j by $\langle\omega\rangle - \omega_i$ (i.e., $\omega_j = (\omega_j + \omega_i) - \langle\omega\rangle$).

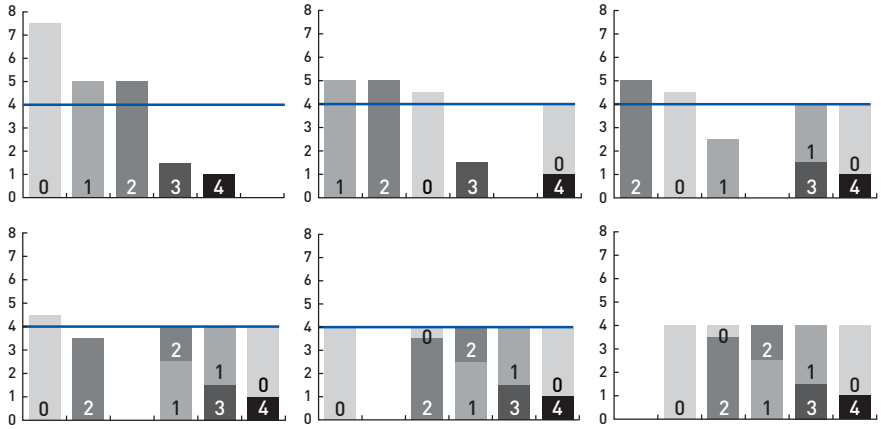


Figure 21-2. *The steps to build an alias table from the discrete distribution in the upper left. The blue line is the average weight.*

Due to the sort, the highest-weighted sample j has above average weight ($\omega_j \geq \langle\omega\rangle$) and the lowest-weighted sample i has $\omega_i \leq \langle\omega\rangle$. This ensures that $\tau \leq 1$ and that only two samples are needed to fill the tuple (because $\omega_i + \omega_j \geq \langle\omega\rangle$). It also ensures that, after creating $\{\tau, i, j\}$, all weight from sample i has been fully accounted for in the table. Sample j's weight gets reduced, as some is included in the newly created tuple.

To see why only two samples are needed per tuple, note that step 4 can reduce ω_j below $\langle\omega\rangle$. In other words, we do not "scrape excess" probability from high-weighted samples into low-weighted ones as if moving dirt with an excavator. We actively ensure that merging two samples gives a complete tuple—even if that leaves the formerly highest-weighted sample with a below-average weight.

Figure 21-2 demonstrates an alias table built from a slightly more complex distribution. Each step creates exactly one alias table tuple, requiring N steps for a distribution with N discrete values. The last step is trivial, as just one sample remains.

21.5 ADDITIONAL READING AND RESOURCES

As a well-known algorithm, many resources discuss the alias method. Many introductory statistics textbooks and Wikipedia provide good starting points. The original paper by Walker [5] is a fairly easy read, and Vose [4] provides an

$O(N)$ table build that observes that you can provide the guarantees outlined in Section 21.4 without any sorting. Devroye [1] presents a clear, more theoretical discussion. Schwarz [3] has an outstanding webpage with clear pseudocode and figures detailing the alias method, Vose's $O(N)$ build, and a discussion of numerical stability.

REFERENCES

[1] Devroye, L. *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986. http://luc.devroye.org/rnbookindex.html.

[2] Pharr, M., Jacob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Practice*. Morgan Kaufmann, 3rd edition, 2016. http://www.pbr-book.org/.

[3] Schwarz, K. Darts, dice, and coins: Sampling from a discrete distribution. https://www.keithschwarz.com/darts-dice-coins/, 2011. Accessed January 6, 2021.

[4] Vose, M. A linear algorithm for generating random numbers with a given distribution. *IEEE Transactions on Software Engineering*, 17(9):972–975, 1991. DOI: 10.1109/32.92917.

[5] Walker, A. J. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software*, 3(3):253–256, 1977. DOI: 10.1145/355744.355749.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 22

WEIGHTED RESERVOIR SAMPLING: RANDOMLY SAMPLING STREAMS

Chris Wyman
NVIDIA

ABSTRACT

Reservoir sampling is a family of algorithms that, given a stream of N elements, randomly select a K-element subset in a single pass. Usually, K is defined as a small constant, but N need not be known in advance.

22.1 INTRODUCTION

We describe *weighted reservoir sampling*, a well-studied algorithm [2, 8] largely unknown to the graphics community. It efficiently samples random elements from a data stream. As real-time ray tracing becomes ubiquitous, GPUs may shift from processing streams of rays or triangles toward streaming random samples, paths, or other stochastic data. In that case, weighted reservoir sampling becomes a vital tool.

After decades of research, reservoir sampling variants exist that optimize different, sometimes conflicting, properties. Input elements can have uniform or nonuniform weights. Outputs can be chosen with or without replacement (i.e., potential duplicates). With various assumptions, we can fast-forward the stream for sublinear running time, minimize the required random entropy, choose elements with greater than 100% probability, or achieve the optimal running time of $O(K + K \log(N/K))$.

Complex variants are beyond the scope of this chapter, but realize that reservoir sampling provides a flexible menu of options depending on your required properties. This rich history exists because many domains process streaming data. Consider managing modern internet traffic, where most participants lack resources to store more than a tiny percent of the stream. Or, for a more historical application, in the 1980s input data was often streamed sequentially off reel-to-reel tape drives.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_22

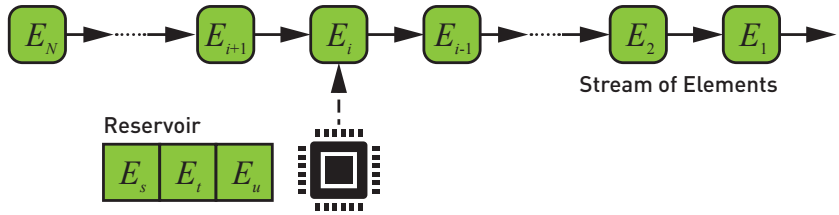


Figure 22-1. *Weighted reservoir sampling processes a stream of elements E_i and incrementally selects a subset weighted proportional to a provided set of weights w_i. Algorithmic variants can sample with or without replacement (i.e., if we guarantee $s \neq t$), can minimize the required random entropy, and can even skip processing all N elements.*

22.2 USAGE IN COMPUTER GRAPHICS

Reservoir sampling has been repeatedly rediscovered in real-time graphics. For instance, it underlies single-pass variations of stochastic transparency [5, 10], which selects random subsets of fragments from a stream of translucent triangles. Lin and Yuksel [6] use a simple variant to trace shadow rays with their desired distribution. And Bitterli et al. [1] use reservoir sampling to stream statistics for spatiotemporal importance resampling.

22.3 PROBLEM DESCRIPTION

Imagine a stream of elements E_i (for $1 \leq i \leq N$), as in Figure 22-1. At a given time, a streaming algorithm processes element i. Elements $j < i$ were previously considered and no longer reside in memory, unless copied to local temporaries. Elements $j > i$ have not yet reached the processor.

Weighted reservoir sampling incrementally builds a reservoir $\mathcal{R}$. This is a randomly chosen K-element subset of previously seen elements (i.e., E_1 to E_{i-1}). For each stream element, E_i is either discarded or inserted into $\mathcal{R}$, potentially replacing existing entries. Replaced samples are forgotten, as if they had never been in the reservoir.

Generally, if E_j has a selection weight w_j, then (for $j < i$) the chance $E_j \in \mathcal{R}$ is proportional to its relative weight $w_j / \sum_{k<i} w_k$. After finishing a stream with total weight $W = \sum_{k \leq N} w_k$, the probability $E_j \in \mathcal{R}$ is proportional to w_j/W. If $K = 1$, the probability is exactly w_j/W.

22.4 RESERVOIR SAMPLING WITH OR WITHOUT REPLACEMENT

For rendering, we generally want independent and identically distributed (abbreviated i.i.d.) samples to ensure unbiasedness. In reservoir sampling,

this means that the chance of selecting E_t (in Figure 22-1) cannot vary based on E_s; if we enforce $s \neq t$, they may no longer be independent. In the reservoir sampling terminology, sampling with replacement clearly gives independence, as any input may occur multiple times in the reservoir.

Fortunately, reservoir sampling with replacement is easier. Delving into the theory literature, most papers thus optimize reservoir sampling without replacement. We leave exploring this literature to the interested reader.

When sampling with replacement, we first focus on understanding the algorithm with reservoir size $K = 1$. Using replacement, samples are i.i.d., so those given by running the algorithm three times with $K = 1$ are distributed identically to those produced by running it once with $K = 3$.

22.5 SIMPLE ALGORITHM FOR SAMPLING WITH REPLACEMENT

For $K = 1$, the reservoir is a tuple $\mathcal{R} = \{R, w_s\}$ containing the currently selected element R and the sum of weights for all previously seen elements $w_s = \sum_{k<i} w_k$. It gets initialized to $\{\emptyset, 0\}$ before processing the stream.

Then, for every stream element E_i with weight $w_i \geq 0$, the reservoir is updated as follows, given a uniform random variate ξ:

```
update(E_i, w_i)
    w_s ← w_s + w_i
    ξ ←rand() ∈ [0...1)
    if (ξ < w_i/w_s)
        R ← E_i
```

When $w_i = 0$, this should leave the reservoir unmodified. This happens naturally, due to IEEE-754 NaN (not a number) behavior, but explicitly checking may be needed to guarantee this behavior for $w_1 = 0$ on non-conformant hardware.

That this update algorithm selects E_i with probability $w_i / \sum_{k \leq N} w_k$ after processing N elements can be easily shown by induction. If $N = 1$, we select E_1 with probability $w_1/w_1 = 1$ (or with zero probability if $w_1 = 0$).

Before processing element E_i at step i, assume that the reservoir contains $\{E_j, \sum_{k<i} w_k\}$, where E_j has been selected with probability $w_j / \sum_{k<i} w_k$. By design, the `update()` function selects E_i with probability

$$\frac{w_i}{w_i + \sum_{k<i} w_k} = \frac{w_i}{\sum_{k \leq i} w_k}. \tag{22.1}$$

Alternatively, it leaves the prior selection, E_j, in the reservoir with probability

$$1 - \frac{w_i}{w_i + \sum_{k<i} w_k} = \frac{(\sum_{k\le i} w_k) - w_i}{\sum_{k\le i} w_k} = \frac{\sum_{k<i} w_k}{\sum_{k\le i} w_k}. \tag{22.2}$$

As E_j was previously selected with probability $w_j/\sum_{k<i} w_k$, its final weighting is

$$\left(\frac{w_j}{\sum_{k<i} w_k}\right)\left(\frac{\sum_{k<i} w_k}{\sum_{k\le i} w_k}\right) = \frac{w_j}{\sum_{k\le i} w_k}. \tag{22.3}$$

That leaves either sample E_i or E_j in the reservoir with the desired probability.

22.6 WEIGHTED RESERVOIR SAMPLING FOR $K > 1$

Extending the algorithm from the previous section for a reservoir with more than one entry (with replacement) is very straightforward. Now the reservoir is $\{\{R_1, \ldots, R_K\}, w_s\}$ and gets initialized to $\{\{\emptyset, \ldots, \emptyset\}, 0\}$. The stream update becomes:

```
update(E_i, w_i)
    w_s ← w_s + w_i
    for (k ∈ 1...K)
        ξ_k ←rand()∈ [0...1]
        if (ξ_k < w_i/w_s)
            R_k ← E_i
```

22.7 AN INTERESTING PROPERTY

A key property of reservoir sampling is that one can combine multiple independent reservoirs without reprocessing their input streams. This is vital in some streaming contexts where it is impossible to replay the streams for a second look.

To combine two reservoirs $\{R_1, w_1\}$ and $\{R_2, w_2\}$, you get $\{R, w_1 + w_2\}$, where $R = R_1$ with probability $w_1/(w_1 + w_2)$; otherwise $R = R_2$.

22.8 ADDITIONAL READING

The Wikipedia page for reservoir sampling [9] is a fine starting point for further reading. The algorithm given here is a simplified "A-Chao" [2] from Wikipedia. Other variants avoid storing the sum w_s, but are less intuitive. Because renderers often need sum-of-weights normalization, storing this sum seems useful. However, the variants that exponentiate random variates may prove useful for volumetric transport.

Much prior research on reservoir sampling has occurred in biostatistics (e.g., Chao [2]) leaving them unaccessible behind paywalls and containing

domain-specific jargon. We found Tillé [7] to be a good statistics reference covering reservoir sampling. We highly recommend the work of Efraimidis and Spirakis, which includes several comparisons and surveys of reservoir variants [3, 4] and is largely comprehensible to non-statisticians.

REFERENCES

[1] Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A., and Jarosz, W. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics*, 39(4):148:1–148:17, 2020. DOI: 10.1145/3386569.3392481.

[2] Chao, M. T. A general purpose unequal probability sampling plan. *Biometrika*, 69(3):653–656, 1982. DOI: 10.2307/2336002.

[3] Efraimidis, P. Weighted random sampling over data streams. *Computing Research Repository (CoRR)*, arXiv, https://arxiv.org/abs/1012.0256, 2010.

[4] Efraimidis, P. and Spirakis, P. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(3):181–185, 2006. DOI: 10.1016/j.ipl.2005.11.003.

[5] Enderton, E., Sintorn, E., Shirley, P., and Lubke, D. Stochastic transparency. In *Symposium on Interactive 3D Graphics and Games*, pages 157–164, 2010. DOI: 10.1145/1730804.1730830.

[6] Lin, D. and Yuksel, C. Real-time rendering with lighting grid hierarchy. In *Symposium on Interactive 3D Graphics and Games*, 8:1–8:10, 2019. DOI: 10.1145/3321361.

[7] Tillé, Y. *Sampling Algorithms*. Springer-Verlog, 2006. DOI: 10.1007/0-387-34240-0.

[8] Vitter, J. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985. DOI: 10.1145/3147.3165.

[9] Wikipedia. Reservoir sampling. https://en.wikipedia.org/wiki/Reservoir_sampling. Accessed January 6, 2021.

[10] Wyman, C. Exploring and expanding the continuum of oit algorithms. In *High Performance Graphics*, pages 1–11, 2016. DOI: 10.2312/hpg.20161187.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 23

RENDERING MANY LIGHTS WITH GRID-BASED RESERVOIRS

Jakub Boksansky, Paula Jukarainen, and Chris Wyman
NVIDIA

ABSTRACT

Efficient rendering of scenes with many lights is a longstanding problem in graphics. Sampling a light to shade from the pool of all lights, e.g., using next event estimation, is a nontrivial task. Sampling must be computationally efficient, must select lights contributing significantly to the shaded point, and must produce low noise while introducing little or no bias. Typically, the light pool is preprocessed to create a data structure that accelerates sampling queries; this may be complex to implement, build, and update.

This chapter builds a new sampling algorithm, *ReGIR*, based on a simple uniform grid structure and the recent screen-space resampling algorithm ReSTIR, which we extend to sample secondary rays in world space.

23.1 INTRODUCTION

Real-time ray tracing enables us to render more realistic images in games than was possible before. Accurate shadows, indirect illumination, and reflections have already been implemented in recent games. In combination with suitable denoisers, we can use ray tracing to render scenes lit by many more lights, without precomputing the static lighting, and with support for dynamic worlds. This is an attractive goal that can differentiate ray traced games from rasterized games in a significant way. Furthermore, we do not have to rely only on analytic lights, as we can also use emissive geometry for lighting.

This is possible by using shadow rays instead of shadow mapping to resolve visibility, avoiding the performance cost of preparing a shadow map for every light source, but also introducing the cost of a denoiser to filter the results. Though shadow rays are a much more flexible approach than shadow mapping, rendered images will be inherently noisy when lit by many different lights.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_23



Figure 23-1. *The Amazon Lumberyard Bistro [1], containing 65,535 emissive triangles, rendered with our ReGIR method using Falcor's [2] path tracer and denoised.*

In this chapter, we focus on a problem of selecting the best light for shading in a way to reduce noise and ensure consistent results.

23.2 PROBLEM STATEMENT

In virtual scenes, there are usually multiple lights simultaneously illuminating each point. (See, for example, Figure 23-1). But when scenes have hundreds or thousands of lights, identifying those actually contributing is a difficult task. Without evaluating the bidirectional reflectance distribution function (BRDF) and visibility of all lights, any one is a potential candidate.

An approach that simply selects a random light without considering visibility gives noisy images, as occluded or distant lights are as likely to be sampled as nearby, visible lights. Importance sampling improves the results (see the pseudocode in Figure 23-2), as the light selection probability varies based on its contribution, selecting lights with higher contribution more often.

Ideally, this probability varies based on the full BRDF and a visibility test; this gives outstanding samples, but is expensive to evaluate (i.e., as expensive as using all lights to shade). Good results are achievable by skipping the expensive visibility test and only evaluating the geometry term (cosine term and distance-based attenuation). This quickly discards backfacing lights and those that are dim or distant. Artists can also design scenes with this in mind,

```
function directLighting(Point p, Direction ω_o)
    result = 0;
    for i ∈ {1 . . . SAMPLE_COUNT} do
        light, pdfLight = SampleLight(p);
        if ShadowRay(p, light) then
            result += Brdf(light, p, ω_o) / pdfLight;
    return (result / SAMPLE_COUNT);
```

Figure 23-2. *Importance sampling loop taking the selected number of light samples at point p, observed from direction ω_o, and averaging them. A basic implementation might select a light randomly from the light pool and set* pdfLight = 1/LIGHTS_COUNT.

using lights with limited emission profiles for quick culling, e.g., spotlights. Because typical distance-based attenuation never reaches zero, modifying attenuation functions to quickly reach zero helps limit light range and further improves performance.

With importance sampling, we can reduce noise by selecting highly contributing lights more often, but basic implementations must still evaluate the probabilities of all the lights to determine which to sample.

23.2.1 RESAMPLED IMPORTANCE SAMPLING

To reduce this cost, we use *resampled importance sampling* (RIS) [5], a powerful technique to numerically sample distributions that are difficult to analytically sample. RIS is a two-step process. First, we select M candidates from the pool of all N samples using a cheap-to-evaluate *source* probability density function (PDF), e.g., uniformly. Then we resample from the M-element subset according to a more expensive *target* PDF. The target PDF may be the ideal distribution discussed previously, or it could include some mixture of BRDF, visibility, geometry, and light contributions. Note that the target PDF is only evaluated for M samples, which is significantly less than N. Resampling can be efficiently implemented using *weighted reservoir sampling* (WRS), described in Chapter 22, and is summarized in Figure 23-3.

Resampling is a key building block of the recently introduced ReSTIR (reservoir spatiotemporal importance resampling) algorithm [3], designed to render scenes with many lights without maintaining a complex data structure. ReSTIR maintains a subset of light samples (called a reservoir) per pixel. The algorithm starts by sampling lights in an inexpensive way, then uses

```
Struct LightSample
    uint lightID;
Struct Reservoir
    LightSample sample;
    uint M;
    float totalWeight;
    float sampleTargetPdf;
function sampleLightRIS(Point p)
    Reservoir r;
    for i ∈ {1 . . . M} do
        candidate, sourcePdf = sampleFromSourcePool();
        targetPdf = PartialBrdf(candidate, p);
        risWeight = targetPdf / sourcePdf;
        r.totalWeight += risWeight;
        r.M++;
        if rand() < (risWeight / r.wSum) then
            r.sample = candidate;
            r.sampleTargetPdf = targetPdf;
    resultWeight = (r.totalWeight / M) / r.sampleTargetPdf;
    return r.sample, resultWeight;
```

Figure 23-3. *Basic RIS using weighted reservoir sampling to sample lights. This example calls* sampleFromSourcePool *to draw samples using an inexpensive method and resamples according to* PartialBrdf *for the shaded point p. The weight of the selected sample* resultWeight *is used as the inverse PDF in importance sampling, during shading. The reservoir structure stores the selected sample and metadata about its construction as explained in Section 23.2.2. The* LightSample *structure simply references a sampled light by its index.*

resampling to spatially and temporally reuse neighbor samples. ReSTIR further improves samples by combining existing reservoirs into new ones, a key component to continuously improve the reservoir in any pixel over many frames.

23.2.2 RESERVOIR

A reservoir, as defined by Bitterli et al. [3], is a data structure that holds one or more samples selected from a larger set (our implementation uses reservoirs of one light sample). A reservoir also stores metadata about how it was constructed, specifically the number of candidates evaluated during its construction M, their total weight, and the target PDF of the selected sample

(see the pseudocode in Figure 23-3). This data is needed for using the reservoir to perform unbiased sampling.

There are multiple ways to construct a reservoir; the first that we will cover is based on the RIS algorithm. Going back to our description of RIS, we can see how they are related: a reservoir is a structure holding a subset of a larger set, and RIS is the algorithm producing such a subset. The reservoir metadata are a byproduct of the resampling process; i.e., the number of candidates *M*, their total weight, and the target PDF of selected samples are all used in the code in Figure 23-3 when running a RIS algorithm, and they can be stored in the reservoir directly.

Another way to create a reservoir is by merging multiple reservoirs into one using the procedure shown in Figure 23-4. As mentioned in the problem statement of this section, this is a key component of ReSTIR and a powerful tool enabling us to construct large numbers of independent reservoirs in parallel using RIS and to merge them to obtain even higher quality sample sets.

The original ReSTIR algorithm works in screen space, meaning it samples lights for pixels on screen and hence only works for primary rays. Our method adapts the ideas in ReSTIR to work in world space. Instead of maintaining a reservoir per pixel, we create a coarse world-space grid and maintain a number of independent reservoirs in each voxel.

```
function updateReservoir(Reservoir result, Reservoir input)
    result.totalWeight += input.weight;
    result.M++;
    if rand() < (input.weight/result.totalWeight) then
        result.sample = input.sample;
        result.sampleTargetPdf = input.sampleTargetPdf;
    return result;
function mergeReservoirs(Reservoir r1, Reservoir r2)
    Reservoir merged;
    updateReservoir(merged, r1);
    updateReservoir(merged, r2);
    merged.M = r1.M + r2.M;
    return merged;
```

Figure 23-4. *A process that merges two reservoirs into one. Note that by calling* updateReservoir *multiple times, we can merge any number of reservoirs.*

23.3 GRID-BASED RESERVOIRS

Our main idea is to split light sampling into two steps. First, we create a pool of samples likely to contribute to a certain area (i.e., the region in a grid cell). In this step, we resample the initial samples according to the grid cell position by estimating light intensity at its area. Here, we can also cull lights that have no contribution to the grid cell. In the second step, we resample according to the BRDF contribution at the shaded point. Both steps produce high-quality samples using RIS to quickly draw samples using a cheap source PDF, but in the second step we sample from the smaller pool. This chains resampling steps, evaluating a more expensive target PDF in the second step but working from the smaller sample count provided by the first step.

23.3.1 SELECTING LIGHT SAMPLES FOR THE GRID

The first step draws samples uniformly from the pool of all lights, a constant time operation, and then resamples according to the target probability based on the light intensity at the grid cell's position (attenuated due to squared distance). It is important to clamp the light distance to the cell borders, otherwise lights inside the cell would be incorrectly prioritized. Note that any method for drawing samples can be used to select initial candidates, e.g., the efficient *alias method* discussed in Chapter 21. Our method works for any light type, as we only evaluate the light intensity for the first step and the BRDF in the second step.

23.3.2 SAMPLING THE LIGHT FOR SHADING

Now, the selected samples in each grid cell can be seen as a pool of single-light reservoirs created by RIS, as shown in Figure 23-5. We can merge these into a single reservoir for shading. Looping over all reservoirs

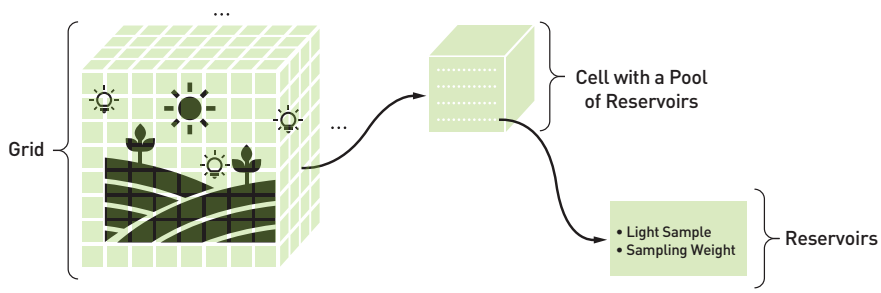


Figure 23-5. *A depiction of grid cells, each storing reservoirs of light samples.*

exhaustively, however, results in poor performance when the number of reservoirs per grid cell is large (we use 512 as a default), and we still need to resample these lights again to account for the BRDF.

To solve both issues, we use the resampling procedure from Figure 23-3 again to merge reservoirs and resample according to the BRDF at the same time. Here, the BRDF is also multiplied by light intensity at the shaded point when calculating the target PDF for RIS. Reservoir merging selects one best light from many reservoirs, whereas resampling implements this merging efficiently, without the need to iterate over all input reservoirs. This time, the resampling target probability is set to the partial BRDF for a given shaded point. The source pool is the set of reservoirs we want to merge. The source PDF is more complex and is described in Section 23.4.2.

Note that the first and second steps are decoupled and do not depend on the screen resolution, world complexity, or number of lights. We can draw any number of lights from the pool in the grid cell once it is built, and we can build pools of lights corresponding to any area in the scene, as we will discuss in Section 23.4.1.

This concludes the high-level description of our algorithm; we cover important implementation details in the next section.

23.4 IMPLEMENTATION

23.4.1 CONSTRUCTION OF THE GRID

To construct our grid, we must first decide on the grid dimensions and how many light samples to store per cell. There will be multiple reservoirs in each grid cell, which we call *light slots*, each storing one light sample. Parameters can be established by experimentation, potentially varying with scene light count, scene extent, visibility range, and desired performance. Our default allocates a grid with 16^3 cells, giving 4096 cells in total. We also default to 512 light slots per grid cell, but depending on the scene lighting complexity, values from 64 to 1024 give good results. As virtual worlds are often flat, allocating a grid with fewer cells along the vertical axis is often advisable.

Using these parameters, we can allocate a GPU buffer of the necessary size. Each light is represented by the `Reservoir` structure, which can use 16-bit precision numbers for all fields except `totalWeight` to reduce memory requirements. As an optimization, we need not explicitly store the number of



Figure 23-6. *Left: our Burger Restaurant scene. Right: an illustration of the world-space grid cell placement. Individual cells are highlighted by random colors.*

seen candidates M in the reservoir, as it is only used to normalize `totalWeight`, which can be precalculated. This also prevents M from growing to infinity as the number of seen samples grows over time. With these optimizations, the memory requirements for the default setting are only 16 MB.

POSITIONING THE GRID

Deciding how to place our grid in the scene depends largely on the rendered content. The simplest approach stretches the grid so that it spans all geometry (see, e.g., the Figure 23-6). This works well for relatively small scenes of known size, but many games feature open-ended worlds with dynamically loaded content, where scene size constantly changes.

For these cases, we recommend one of two approaches. A scrolling clipmap can be used, which ensures that individual grid cells seemingly stay in place and their positions are given by the window into a clipmap centered around the camera. In this case, the range of the grid (or size of the cell) has to be determined, which prevents our light sampling from being used outside of this range.

Another possible approach is to implement a sparse grid, where the positions of individual cells are determined using a hash map. In this case, we map each point in world space to a grid cell in the hypothetical infinite uniform grid as:

```
1     return int3(worldCoords / gridCellSize);
```

Once we try to sample from a grid cell at given coordinates, we store these coordinates into the map, at a position determined by the hash. Note that conflicts must be resolved at this point. When we construct the grid, we first

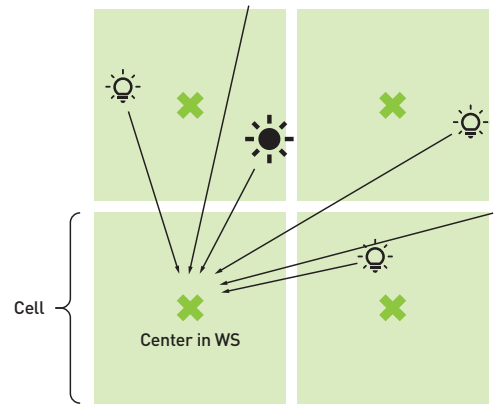


Figure 23-7. *Construction of the grid. RIS selects a number of candidates and resamples them according to the light intensity at the cell center.*

read the map to determine which cells in the world space are used, map them back to world space, and create reservoirs only for them.

BUILDING CELL RESERVOIRS

To fill the grid with light samples, we apply the RIS algorithm (see Figure 23-3) for each entry (light slot) in a grid. Remember that each grid cell contains multiple light slots, i.e., multiple light samples that were created by running the RIS algorithm.

Because selecting a per-slot light sample is done independently of other slots, even from the same cell, sampling can be parallelized, making grid construction very fast. This procedure first maps thread ID to the corresponding grid cell (see Figure 23-7), which is in turn used to map the cell position to world space:

```
// Calculate the grid cell coordinates.
int gridCellLinearIndex = threadID / gGridLightsPerCell;
int3 cellCenterCoords = linearToCoords3D(gridCellLinearIndex);

// Calculate the grid cell center in world coordinates.
float3 gridCellCenter = mapGridToWorld(cellCenterCoords);

// Select a light to be stored in evaluated light slot.
Reservoir lightSlotReservoir = sampleLightRIS(gridCellCenter);
```

By default, the number of initial RIS candidates (M) we use is eight. Increasing it quickly encounters the law of diminishing returns. As mentioned in

Section 23.3.1, the target PDF in `sampleLightUsingRIS` only depends on the intensity of the light at the grid center, as follows. Note the clamping that ensures that lights inside the cell have the same probability:

```
float3 lightVector = candidate.position - gridCellCenter;
float lightDistanceSquared = max(gMinDistanceSquared, dot(lightVector,
    lightVector));
float sourcePdf = <Source PDF as described, e.g., uniform sampling>;
float targetPdf = sample.intensity / lightDistanceSquared;
```

This code assumes a typical distance-based attenuation function for all lights, i.e., attenuating intensity with the distance squared, but any type of light can be supported. Most notably, directional lighting from the sun has the same intensity everywhere in the scene.

To ensure that each reservoir's running count *M* of the light candidates that it incorporates does not grow to infinity, we normalize it before storing the reservoir in the grid. This also enables removing *M* from the reservoir, as it is always one at this point:

```
lightSlotReservoir.totalWeight = lightSlotReservoir.totalWeight /
    lightSlotReservoir.M;
lightSlotReservoir.M = 1;
```

Note that using this process, a light can end up in *any* grid cell where it has a nonzero probability of contribution. Thus, our grid is not a spatial subdivision structure, but rather a data structure to store samples and their probabilities.

Finally, an important optimization is to only update grid cells that are used. Cells covering empty space will never be used for shading, so we can skip filling their light slots. We implement a cache where each cell stores a frame number for when it was last accessed. This information is cached by the sampling routine. During per-frame construction, we first check whether each cell has been recently accessed (e.g., in the last eight frames) and only update cells in active use.

TEMPORAL REUSE

The construction process just described is repeated each frame, rebuilding the grid from scratch. However, we can use reservoirs of lights from previous frames to continuously improve the grid in the most recent frame using the reservoir merging process described in Section 23.3.2. This is an important technique also used in the original ReSTIR implementation [3] that achieves much more stable results.

To handle temporal reuse, we also retain grids from previous frames (eight by default). During grid construction, we compute each reservoir as previously described, but before storing it into the grid, we merge this new reservoir (see Figure 23-4) with the reservoirs from prior frames (corresponding to the same light slot):

```
// Merge new reservoir with reservoirs from previous frames.
for (int i = 0; i < GRIDS_HISTORY_LENGTH; i++) {
    lightSlotReservoir = mergeReservoirs(lightSlotReservoir,
        gLightGridHistory[i][lightSlotIndex]);

    // Divide wSum by M after each combining "round."
    lightSlotReservoir.totalWeight /= lightSlotReservoir.M;
    lightSlotReservoir.M = 1;
}
```

Because good light samples have higher weight, it is likely that we keep reusing them until even better samples are found. This helps to continuously improve our samples.

DYNAMIC LIGHTS

Our method supports dynamic lights out of the box. We have intentionally used a light ID in the reservoir structure to reference sampled lights. This requires indirection to access light properties, but also ensures that every time we use a light, its current properties are used for shading. However, a problem can occur with temporal reuse, which does not modify a light's weight even if it changes. This can introduce excessive noise if the lights change significantly. As a workaround, we can prevent dynamic lights from participating in temporal reuse, so they will be replaced by a new sample. Alternatively, a more expensive reservoir merge can re-weigh dynamic lights during grid construction.

23.4.2 SAMPLING FROM THE GRID

When sampling lights, we start by finding the grid cell corresponding to the shaded point (see Figure 23-8). To reduce artifacts from nearest neighbor lookups (i.e., visible cell boundaries), we first randomize the lookup with an offset proportional to the cell size. This essentially performs stochastic trilinear filtering, sampling neighboring cells proportionally to their distance from the lookup:

```
// Jitter hit point position within the size of grid cell.
float3 gridLoadPosition = pointPosition+(float3(rand(), rand(), rand()) *
     2.0f - 1.0f) * gHalfGridCellSize;

```

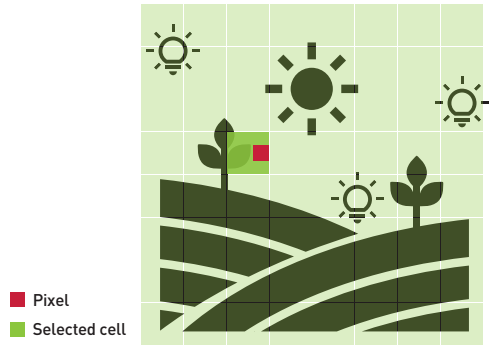


Figure 23-8. *Sampling from the grid. First, a cell in the neighborhood of the pixel is selected, then RIS is used to resample lights stored in the cell at the pixel. Light sources indicated in the image are an example of lights stored in sampled grid cells.*

```
// Figure out which grid cell to sample from (gridCellStartIndex) based on
    the jittered hit position.
int3 gridCellCoords = mapWorldToGrid(gridLoadPosition);
uint gridCellIndex = coords3DToLinear(gridCellCoords);
uint gridCellStartIndex = gridCellIndex * gGridLightsPerCell;
```

This gives us `gridCellStartIndex`, a position in the grid buffer where the grid cell's pool of samples starts. Next, as discussed in Section 23.3.2, we apply RIS again to merge reservoirs in the pool into a final light sample for shading, and also we resample according to the target PDF (which is based on the BRDF at the shaded point). Here, RIS draws source samples directly from the reservoirs of the selected grid cell:

```
float sourcePdf = candidate.sampleTargetPdf / reservoirAverageWeight;
float targetPdf = PartialBrdf(candidate, shadedPoint);
```

We calculate the source probability as the target PDF of candidates from the first RIS pass (during grid construction) divided by the average weight of all reservoirs (light slots) in the grid cell. This is an iterative application of RIS, similar to the approach described in the original ReSTIR article [3]. This average may be used multiple times, so we precalculate it for each cell in a separate pass. The function `PartialBrdf` depends on renderer-specific BRDFs and light parameters and can be a full BRDF if the cost is reasonable.

Finally, a pixel can be shaded using the light sample and the RIS weight:

```
Reservoir risReservoir = sampleLightUsingRIS(shadedPoint);
float lightSampleWeight = (risReservoir.totalWeight / M) / risReservoir.
    sampleTargetPdf;
LightSample light = loadLight(risReservoir.lightSample);
return light.intensity * Brdf(light, shadedPoint) * lightSampleWeight;
```

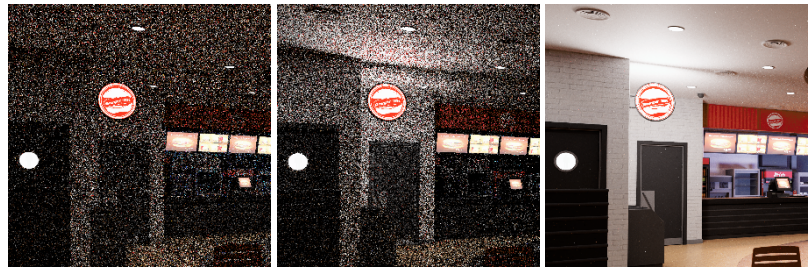


Figure 23-9. *A single frame rendered using naive uniform sampling (left) and ReGIR (center) and the accumulated result (right) of the Burger Restaurant scene. Note the significantly better sampling around the light source in top left corner using our method.*

23.5 RESULTS

We implemented and evaluated ReGIR (our method) in the Falcor rendering framework [2]. Compared to naive uniform sampling and basic RIS, we achieved superior results (see Figures 23-9 and 23-10) with only a small per-frame performance cost of about 1.5 ms using our default settings. Performance was measured with a RTX 3090 GPU and 1920 $\times$ 1080 resolution

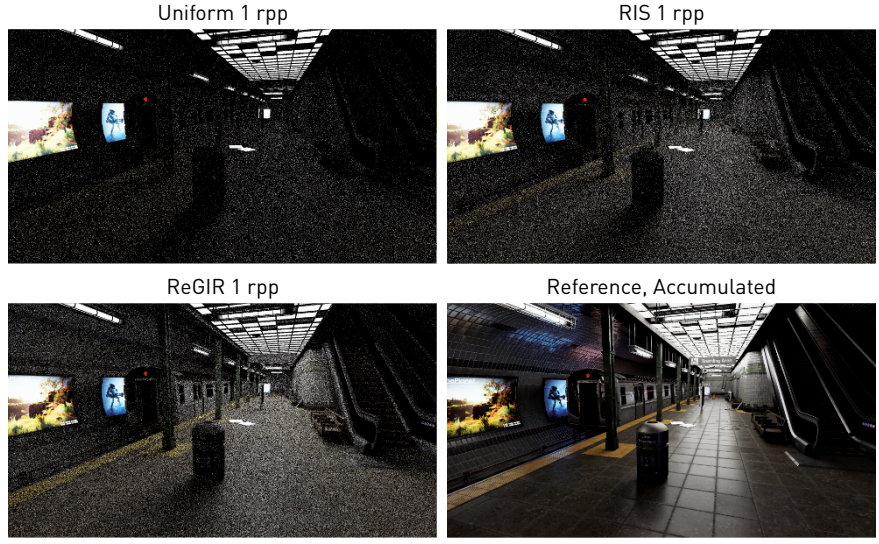


Figure 23-10. *Comparison of naive uniform sampling (top left) to only using RIS with 16 candidates (top right), our ReGIR method (bottom left), and a ground truth (bottom right). We use one ray per pixel (rpp).*

and is similar across most tested scenes. Of this cost, about 0.3 ms is spent on constructing the grid. The remaining cost comes from selecting lights from the grid for shading; this cost depends on the resolution and number of secondary rays accessing the grid. Our performance depends mostly on how many light samples we store in the grid and the number of nonempty cells due to the scene complexity.

Visual quality depends on the number of light samples that we can afford to store per grid cell, and on the size of the cells in world space. Using smaller cells can improve visual quality, but can also limit the range of the grid.

We show results of our method applied to primary rays; however, its expected use case is shading secondary rays and arbitrary points in the scene, whereas screen-space ReSTIR typically gives better quality on primary hits.

The slight bias of our method can be attributed to the discrete nature of the grid and the limited number of samples stored in each grid cell. Temporal reuse can also contribute to the bias. In real-time applications, this should not pose significant issues as we believe high performance is preferable, and the presence of a denoiser should smooth out any remaining artifacts.

23.6 CONCLUSIONS

In this chapter, we presented our new ReGIR algorithm for many light sampling, which can be used in real-time ray tracing for both primary and secondary rays. Because it uses a simple data structure, it is relatively easy to implement in game engines and provides high performance, although other, more costly methods can give superior results in terms of lower noise. However, when using a specialized denoiser, higher performance can be a more important benefit, assuming input noise is low enough to produce sharp and stable results after denoising.

Because our world-space method uses a larger granularity than screen-space ReSTIR, it is not as well suited for shading primary ray hits, except in scenes with only a few lights. However, it enables shading secondary hits, and combining these two methods might be a preferred way of using ReSTIR. ReGIR can also be combined with methods for calculating global lighting such as dynamic diffuse global illumination (DDGI) [4] or with a path tracer.

REFERENCES

[1] Amazon Lumberyard. Amazon Lumberyard Bistro. *Open Research Content Archive (ORCA)*, http://developer.nvidia.com/orca/amazon-lumberyard-bistro, 2017.

[2] Benty, N., Yao, K.-H., Clarberg, P., Chen, L., Kallweit, S., Foley, T., Oakes, M., Lavelle, C., and Wyman, C. The Falcor real-time rendering framework. https://github.com/NVIDIAGameWorks/Falcor, 2020. Accessed August 2020.

[3] Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A., and Jarosz, W. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 39(4):148:1–148:17, July 2020. DOI: 10/gg8xc7.

[4] Majercik, Z., Guertin, J.-P., Nowrouzezahrai, D., and McGuire, M. Dynamic diffuse global illumination with ray-traced irradiance fields. *Journal of Computer Graphics Techniques (JCGT)*, 8(2):1–30, 2019. http://jcgt.org/published/0008/02/01/.

[5] Talbot, J., Cline, D., and Egbert, P. Importance resampling for global illumination. In *Eurographics Symposium on Rendering (2005)*, pages 139–146, 2005. DOI: 10.2312/EGWR/EGSR05/139-146.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 24

USING BLUE NOISE FOR RAY TRACED SOFT SHADOWS

Alan Wolfe
NVIDIA

ABSTRACT

Ray tracing is nearly synonymous with noise. Importance sampling and low-discrepancy sequences can help reduce noise by converging more quickly, and denoisers can remove noise after the fact, but sometimes we still cannot afford the number of rays needed to get the results we want. If using more rays is not an option, the next best thing is to make the noise harder to see and easier to remove. This chapter talks about using blue noise toward those goals in the context of ray traced soft shadows, but the concepts presented translate to nearly all ray tracing techniques.

24.1 INTRODUCTION

In these early days of hardware-accelerated ray tracing, our total ray budgets are as low as they are ever going to be. Even as budgets increase, the rays are going to be eaten up by newer techniques that push rendering even further. Low sample count ray tracing is going to be a topical discussion for a long time.

Though many techniques exist to make the most out of every ray—such as importance sampling and low-discrepancy sequences—blue noise is unique in that the goal is not to reduce noise, but to make it harder to see and easier to remove.

Blue noise is a cousin to low-discrepancy sequences because they both aim to spread sample points uniformly in the sampling domain. This is in contrast to white noise, which has clumps and voids (Figure 24-1). Clumped samples give redundant data, and voids are missing data. Blue noise avoid clumps and voids by being roughly uniform in space.

Being blue noise or low discrepancy is not mutually exclusive, but this chapter is going to focus on blue noise. For a more thorough understanding of

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_24

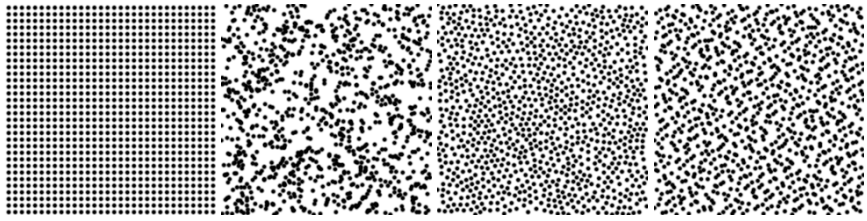


Figure 24-1. *Arrays of 1024 samples. Left to right: regular grid, white noise, blue noise, and the Halton sequence. Clumps of samples are redundant; gaps are missing information. Patterns can cause aliasing.*

modern sampling techniques, low-discrepancy sequences should be considered as well.

You may think that a regular grid would give good coverage over the sampling domain, but the diagonal distances between points on a grid are about 40% longer than non-diagonal distances, which makes it anisotropic, giving different results based on the orientation of what is being sampled. If you were to address that problem, you would get a hexagonal lattice, like a honeycomb. Using that for sampling, you could still have aliasing problems, though, and convergence may not be much better than a regular grid, depending on what you are sampling.

Blue noise converges at about the same rate as white noise, but has lower starting error. See Figure 24-2 for convergence rates. The power of blue noise is not in removing error, but making the error harder to see and easier to

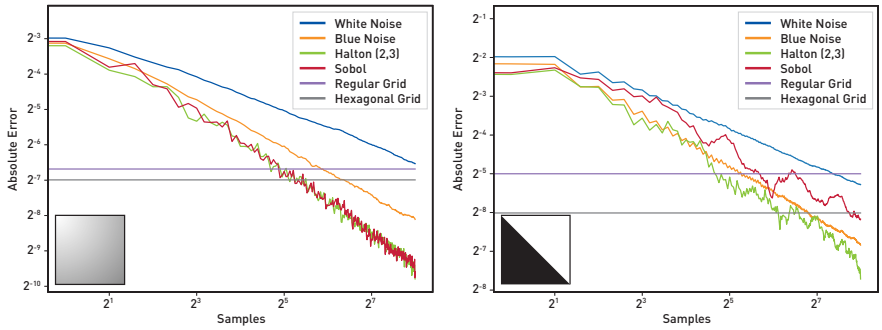


Figure 24-2. *Integrating two functions with 256 samples, with absolute error averaged over 1000 runs. Left: a crop of a Gaussian function. Right: a diagonal step function. Regular and hexagonal grids are not progressive, so only the final error is graphed.*

remove, compared to other sequences. It does this by having low aliasing and having randomization (noise) only in high frequencies.

24.2 OVERVIEW

We are going to go into more detail about blue noise samples in Section 24.3 and blue noise masks in Section 24.4. Then, we are going to look at blue noise from a digital signal processing perspective in Section 24.6.

After that, we are going to explore how to use both blue noise sampling and blue noise masks for ray traced soft shadows in Section 24.7. As a comparison, we will show an alternate noise pattern called interleaved gradient noise (IGN) presented by Jimenez [21], which is specifically designed for use with temporal antialiasing (TAA) [23] in Section 24.8, and then look at all of these things using the ꟻLIP perceptual error metric in Section 24.9.

24.3 BLUE NOISE SAMPLES

Two-dimensional blue noise samples are points in a square that are randomized but fill the square well, and are roughly uniform (often toroidally), as we see in Figure 24-3.

Samples can be blue noise distributed in other domains, too. On a sphere, blue noise would be points that covered the entire surface of the sphere and were randomized, but also were roughly uniform. These points on a sphere can also be viewed as blue noise distributed unit vectors. The similarity metric between vectors would be the negative dot product between those vectors instead of the distance between points on the surface of the sphere. Both perspectives would give equivalent samples, but show how blue noise conceptually can go beyond 2D points in a square.

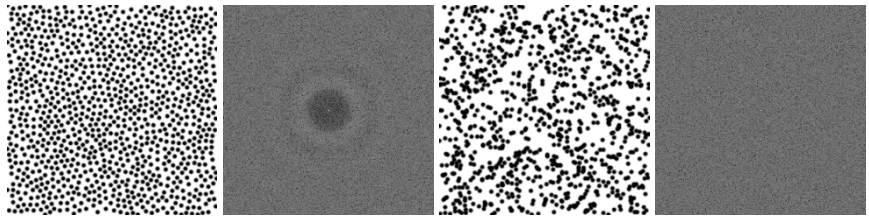


Figure 24-3. *For 1024 dots, blue noise samples (left) and discrete Fourier transform (DFT; middle left) showing attenuated low frequencies, and white noise samples (middle right) and DFT (right) showing all frequency content.*

```
samples = [];
for sampleIndex ∈ [0, NumSamples) do
    bestScore = -∞;
    bestCandidate = {};
    for candidateIndex ∈ [0, sampleIndex + 1) do
        score = ∞;
        candidate = GenerateRandomCandidate();
        for testSample ∈ samples do
            testScore = ToroidalDistance(candidate, testSample);
            score = min(score, testScore);
        if score > bestScore then
            bestScore = score;
            bestCandidate = candidate;
    samples.add(bestCandidate);
```

Figure 24-4. *Mitchell's best-candidate algorithm. `ToroidalDistance()` is the similarity metric being used.*

Generally speaking, blue noise samples can be thought of as blue noise in the form $\mathbf{v} = f(N)$, where N is an integer index into the samples and $\mathbf{v}$ could be a scalar, a point, a vector, or anything else in any sampling domain for which you could define a similarity metric.

Mitchell's best-candidate algorithm [15] can generate high-quality blue noise samples. The algorithm only requires that you are able to generate uniform random samples and have a similarity metric for pairs of samples. See the pseudocode in Figure 24-4. A more modern algorithm can be found in de Goes et al. [5]

Generating blue noise sample points is often computationally expensive. For this reason, blue noise sample points are usually generated in advance, and used as constants at runtime. Because blue noise is about squeezing the most quality out of visuals that you can, precalculating the samples lets you spend more time making better samples in advance, and then having an inexpensive runtime cost. A quicker dart throwing algorithm is compared to Mitchell's best-candidate blue in Figure 24-5, showing that the quicker algorithm does not give equivalent results.

Mitchell's best-candidate algorithm generates progressive blue noise, which means that any length of samples starting at index 0 are blue noise.

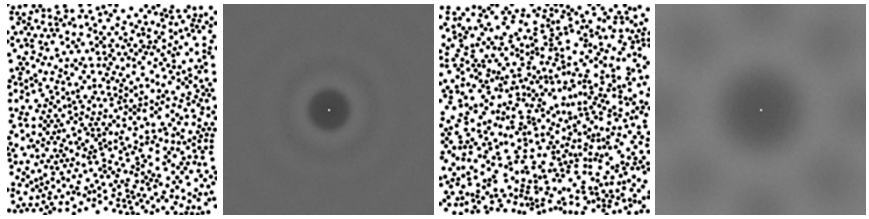


Figure 24-5. *For 1024 samples and 100 averaged DFTs, best-candidate samples (left) and dart throwing samples (right). Dart throwing samples look comparable to best-candidate samples (left half), but frequency analysis (right half) shows that they are significantly different.*

The more modern algorithms usually make non-progressive samples, which mean needing to use all samples before they are blue noise. You can make non-progressive samples progressive after they are generated, though, and the void and cluster algorithm [22] in the Section 24.5 does this as part of its work.

Progressive samples are useful when you do not know how many samples you want to take at runtime—e.g., if you want to take more samples where there is more variance. With progressive blue noise samples, you can generate some number of samples, then choose how many you want to use at runtime. You do not need to generate blue noise samples for every amount you may possibly want to sample, only the maximum amount.

An improved type of blue noise is found in Reinert et al. [19] called projective blue noise. If you generate 2D blue noise and look at the *x*- or *y*-values by themselves, they will look like white noise. If you were using your blue noise to sample a shadow cast by a vertical wall, or nearly vertical wall, the *y*-values of your samples would end up not mattering much to the result of your sampling, so that only the *x*-values mattered. If your *x*-axis looks like white noise, you are not going to do good sampling. Projective blue noise addresses this by making it so that blue noise samples are also blue on all axis-aligned subspace projections.

A nice property of blue noise is that it tends to keep its desirable properties better than other sequences when undergoing transformations like importance sampling [13]. That can be interesting when you have hybrid sampling types, such as the Sobol sequence that has blue noise projections [16].

There has been other progress on combining the integration speed of low-discrepancy sequences, while keeping the pleasing error pattern from blue noise in "Low-Discrepancy Blue Noise Sampling" [1], "A Low-Discrepancy Sampler That Distributes Monte Carlo Errors as a Blue Noise in Screen Space" [10], "Screen-Space Blue-Noise Diffusion of Monte Carlo Sampling Error via Hierarchical Ordering of Pixels" [2], "Orthogonal Array Sampling for Monte Carlo Rendering" [12], and "Progressive Multi-jittered Sample Sequences" [4].

24.4 BLUE NOISE MASKS

The other type of blue noise is blue noise masks, which are commonly referred to as blue noise textures. Blue noise textures are images where neighboring pixels are very different from each other. You can see a blue noise mask in Figure 24-6.

The most common type of blue noise mask is a 2D image with a single value stored at each pixel, but that is not the only way it can be done. You could have a 3D volume texture that stores a unit vector at each pixel, for instance, or any other sort of data on a grid you can imagine.

More generally, blue noise masks can be thought of as blue noise in the form $\mathbf{v} = f(\mathbf{u})$, where $\mathbf{u}$ is some discrete value of any dimension, and $\mathbf{v}$ could be a scalar, a point, a vector, or similar.

Generating blue noise of this form can be done using the void and cluster algorithm [22], which is detailed in Section 24.5. You can also download a zip file of textures made with the void and cluster method from the Internet [17]. How blue noise textures are usually used is that the textures are generated in advance, shipped as regular texture assets, and read by shaders at runtime. This is because, just like blue noise samples, blue noise textures are

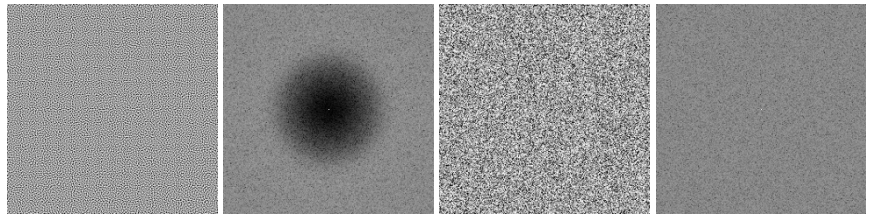


Figure 24-6. *Blue noise mask (left) and DFT (middle left) showing attenuated low frequencies. White noise mask (middle right) and DFT (right) showing all frequency content.*

computationally expensive to generate and can be made higher-quality when precomputed, giving high quality with low overhead at runtime.

There are other algorithms for generating blue noise masks. In Georgiev and Fajardo [7], a texture is initialized to white noise and pixel values are swapped if it reduces the energy function of the texture. Simulated annealing is used to help reach a better local optimum. This algorithm allows for vector-valued blue noise masks, unlike the void and cluster method, which is limited to scalar values.

Another way to make a blue noise mask is to start with white noise and high pass filter it. If you filter white noise, the histogram becomes uneven, though, and if you fix the histogram, it damages the frequency content. A decent way to get around this problem is to do several iterations of filtering the noise and fixing the histogram.

Using these alternate techniques, you can end up with blue noise that looks correct both as a texture and in frequency space, but they both lack an important detail that the void and cluster algorithm has.

Void and cluster blue noise masks have the useful property that thresholding the texture at 10% leaves 10% of the pixels remaining, and those pixels will be in a blue noise sample pattern. That is true for any threshold level. This can be useful if you want to use a blue noise texture for something like stochastic transparency because it will make the pixels that survive the alpha test be in a blue noise pattern.

To use a blue noise mask, the blue noise texture is tiled on the screen and used as the source of per-pixel random numbers. Doing this, the random numbers used by each pixel are going to be very different from neighboring pixels, producing very different results from neighbors, which is blue noise. Blue noise tiles well due to not having any low-frequency structure for your eye to pick up.

An example use of blue noise masks is when quantizing data to lower bit depths. If you add noise before quantization, it turns banding artifacts into noise, which looks more correct. Using blue noise instead of white noise means that the noise is harder to notice, and easier to remove. This can be useful when bringing high dynamic range (HDR) values from rendering into smaller bit depths for display. This can also be useful when quantizing data to G-buffer fields including color data like albedo, as well as non-color data such as normals.

Kopf et al. [14] has a technique that allows you to zoom in or out of blue noise, keeping the noise consistent hierarchically.

Some other details of blue noise masks and their usage are looked at in "The Rendering of INSIDE: High Fidelity, Low Complexity" [8], including triangular distributed blue noise that makes error patterns be more independent of signal, animating blue noise and dealing with blue noise dithering at the low and high ends of values to avoid a problem at the clamping zones.

24.5 VOID AND CLUSTER ALGORITHM

The void and cluster algorithm generates blue noise masks with the additional property that thresholding them causes the surviving pixels to be distributed as blue noise sample points. The algorithm is made up of the following steps:

1. Initial binary pattern.
2. Phase I: Make pattern progressive.
3. Phase II: First half of pixels.
4. Phase III: Second half of pixels.
5. Finalize texture.

The algorithm requires storage for each pixel to remember whether a pixel has been turned on or not, and an integer ordering value for that pixel. All pixels are initialized to being turned off and having an invalid ordering value.

The concept of voids and clusters are used to find where to add or remove points during the algorithm. Voids (gaps) are the low-energy areas in the image, and clusters are the high-energy areas.

Every pixel $\mathbf{p} = (p_x, p_y)$ that is turned on gives energy to every point $\mathbf{q}$ in the energy field using

$$E(\mathbf{p}, \mathbf{q}) = \exp\left(-\frac{\|\mathbf{p} - \mathbf{q}\|^2}{2\sigma^2}\right), \tag{24.1}$$

where $\mathbf{p}$ and $\mathbf{q}$ are the integer coordinates and distances are computed on wrapped boundaries, i.e., *toroidal* wrapping. The σ is a tunable parameter that controls energy falloff over distance, and thus frequency content.
Ulichney [22] recommends $\sigma = 1.5$. That equation may look familiar—it is just a Gaussian blur!

The algorithm can be sped up beyond a naive implementation by calculating the energy field using multithreading—calculating the energy for all pixels is an $O(N^4)$ operation where N is the side length of the output texture. It can be further sped up by actually using a texture to store the energy field as a quick lookup table, where updates are done to it iteratively as pixels are turned on or turned off. Lastly, though technically all pixels should be able to gain energy from all other pixels, in practice there is a radius at which the amount of energy gained is so negligible that it can be ignored. This means that based on the σ used in the Gaussian, you can calculate a pixel radius that contains the desired proportion of the Gaussian's energy. This lets you consider fewer pixels when calculating energy in the energy field. See the following equation for calculating the size (diameter) of the kernel for a given σ, where t is the threshold and a t value of 0.005 means all but 0.5% of the energy is accounted for:

$$\left\lfloor 1 + 2 * \sqrt{-2\sigma^2 \ln t} \right\rfloor . \tag{24.2}$$

24.5.1 INITIAL BINARY PATTERN

The first step is to generate an initial *binary* pattern where not more than half of the pixels are turned on. This can be done using white noise or nearly any other pattern. These pixels need to be transformed into blue noise points before continuing. That is done by repeatedly turning off the largest-valued cluster pixel and turning on the lowest-valued void pixel. This process is repeated until the same pixel is found for both operations. At that point, the algorithm has converged and the initial binary pattern is blue noise distributed.

The reason not more than half of the pixels should not be turned on in this step is because there is different logic required when processing the second half of the pixels compared to the first half. The logic for the first half of the pixels works better for sparser points, whereas the logic for the second half works better for denser points.

24.5.2 PHASE I: MAKE PATTERN PROGRESSIVE

The initial binary pattern is now blue noise distributed, but has no ordering and must be made into a progressive blue noise sequence. This is done by repeatedly removing the highest-energy pixel, i.e., the largest *cluster*, and giving that pixel an ordering of how many pixels are on after it is turned off.

When this is done, the pixels in the initial binary pattern are turned back on. At this point, the initial binary pattern pixels have an ordering that makes them an ordered (progressive) blue noise sampling sequence and we are ready for phase II.

24.5.3 PHASE II: FIRST HALF OF PIXELS

Next, pixels are turned on, one at a time, until half of the pixels are turned on. This is done by finding the lowest-energy pixel, i.e., the smallest *void*, and turning that pixel on. The ordering given to that pixel is the number of pixels that were on before it was turned on.

24.5.4 PHASE III: SECOND HALF OF PIXELS

At this point, the states of all the pixels are reversed. Pixels that are on are turned off, and vice versa. This phase repeatedly finds the largest-valued cluster and turns it off, giving it the ordering of the number of pixels that were off before it was turned off. When there are no more pixels turned off, this phase is finished and all pixels have an ordering.

24.5.5 FINALIZE TEXTURE

After all pixels are ordered, that ordering is turned into pixel values in the output image. If the output image has a resolution of 256×256 pixels, then their ordering will go from 0 to 65,535. If the output is an 8-bit image, the values will need to be remapped from 0 to 255. This will create non-unique values in the output texture, but there will be an equal number of each pixel value—256 of each.

You now have a high-quality blue noise mask texture ready for use!

24.6 BLUE NOISE FILTERING

When stopping before convergence in rendering, there is going to be error left behind in the image. If using a randomized sampling pattern like white noise or blue noise, the error will be left as a randomized noise pattern, and usually you want to remove it as much as possible.

The most common way to remove noise is to blur the image. In digital signal processing terms, a blur is a low pass filter, meaning it lets low frequencies through the filter, while reducing or removing high-frequency details.

Blurring is most commonly done by convolving an image against a blur kernel. This is mathematically the same as if we took the image and the blur

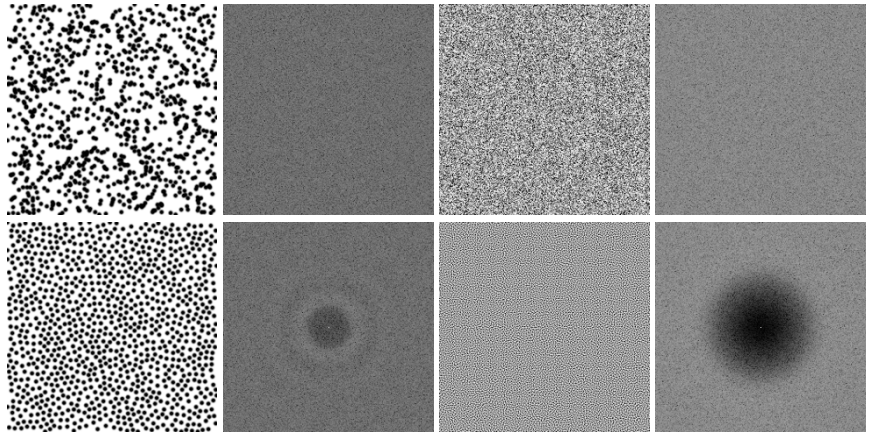


Figure 24-7. *Noise and DFT for noise samples and masks. Top: white noise. Bottom: blue noise.*

kernel into frequency space, multiplied them together, and brought the result out of frequency space again.

In Figure 24-7, you can see how white noise is present in all frequencies, while blue noise is only present in higher frequencies. In Figure 24-8 you can see how various blur kernels look in frequency space. When blurring noise, these are the things getting multiplied together in frequency space.

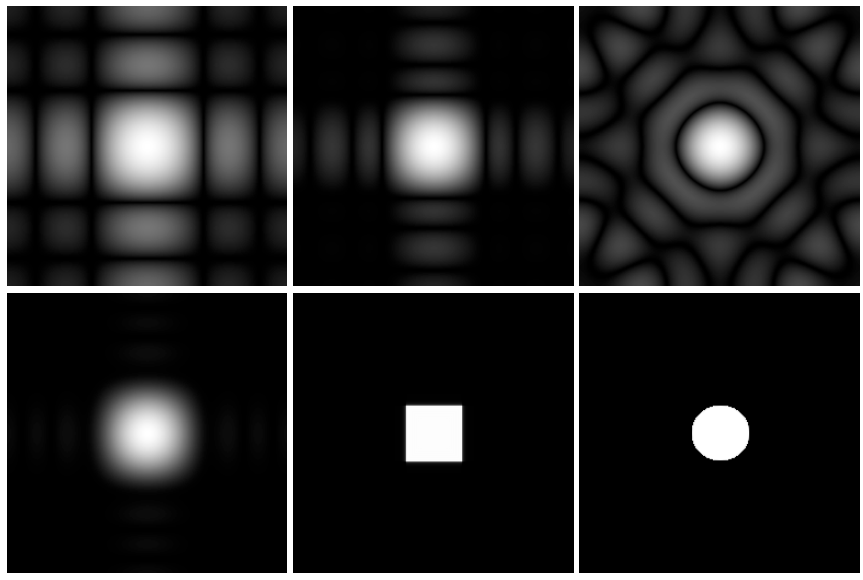


Figure 24-8. *DFT of various blur or low pass filter kernels. Clockwise from upper left: box, a trous, disk, circular sinc, sinc, and Gaussian.*

You can see how multiplying the white noise by any of the kernels is going to leave something that looks like white noise but is in the shape of the filter used, which is going to leave noise of those frequencies behind in the image. The only way you would be able to get rid of white noise completely is by removing all frequencies, which would leave nothing of your render.

Alternatively, you can see how if you were to multiply blue noise in frequency space by the Gaussian kernel or a sinc kernel, the result will be noiseless because the place where the blur kernel is white, the blue noise is black. There is no overlap, so the blue noise will have gone away.

Noisy renders are a mix of data and noise, so when you do a low pass filter to remove the blue noise from the result, it will also remove or reduce any data that is also in the frequency ranges that the blue noise occupies, but this is a big improvement over white noise, which is present in all frequencies.

From a digital signal processing perspective, the ideal isotropic low pass filter in 2D is the circular sinc filter, but from the blue noise filtering perspective, the ideal kernel will look like the missing part of the blue noise frequencies, so would be bright in the center where the low frequencies are and then fade out circularly to the edge of where the blue noise has full amplitude. Looking at these kernels, it seems that a Gaussian blur is the best choice.

If using the other blurs, you may keep frequencies that you did not intend to keep, or you may remove frequencies that you did not need to remove from the render. When doing a Gaussian blur, you want to choose σ such that the filter in frequency space is roughly the same size as the frequency hole in your blue noise. If it is too small, it will leave some blue noise residue, which will look like blobs, and if it is too large, it will remove more of the details than is needed. You can also choose to leave some blue noise residue behind if that preserves sharper details that you care about.

24.7 BLUE NOISE FOR SOFT SHADOWS

Now it is finally time to put all this together into a rendering technique!

24.7.1 LIGHTS AND SHADOWS

When ray tracing shadows, you use ray tracing to answer the question of whether a specific point on a surface can see a specific point on a light. If it can, you add the lighting contribution to the surface.

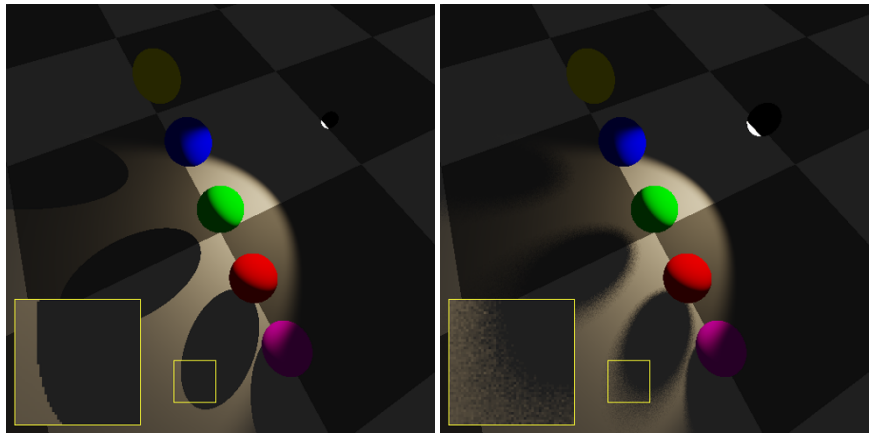


Figure 24-9. *Left: discrete light shadows are noiseless but hard at one ray per pixel. Right: area lights are soft but noisy in the penumbra at 16 white noise samples per pixel.*

Classically, the most common type of light in real-time graphics has been discrete lights, which are infinitely small and come in the usual flavors of positional lights, directional lights, and spotlights. When checking to see if a point on a surface can see a light, there is only one single ray direction toward which to shoot a ray and get a Boolean answer of yes or no. For this reason, when you ray trace shadows for discrete lights, they are completely noiseless at one sample per pixel, but they have hard shadow edges as you can see in Figure 24-9, left.

In more recent times, real-time graphics has moved away from discrete lights toward area lights, which have area and volume. These lights also come in the usual flavors of positional lights, directional lights, and spotlights, but also have others such as capsule lights, line lights, mesh lights, and even texture-based lights, to give variety and added realism to lighting conditions. Because these lights have volume, handling shadows is not as straightforward. The answer is no longer a Boolean visibility value but is more complex because you need to integrate every visible part of the light with the surface's BRDF. Not only will the surface react to the light differently at different angles, the light may shine different colors and intensities in different locations on the light, or in different directions from the light.

The correct way to handle this is discussed by Heitz et al. [11], but we are going to simplify the problem and use ray tracing to tell us the percentage of the light that can be seen by a point on a surface, then use that as a shadow

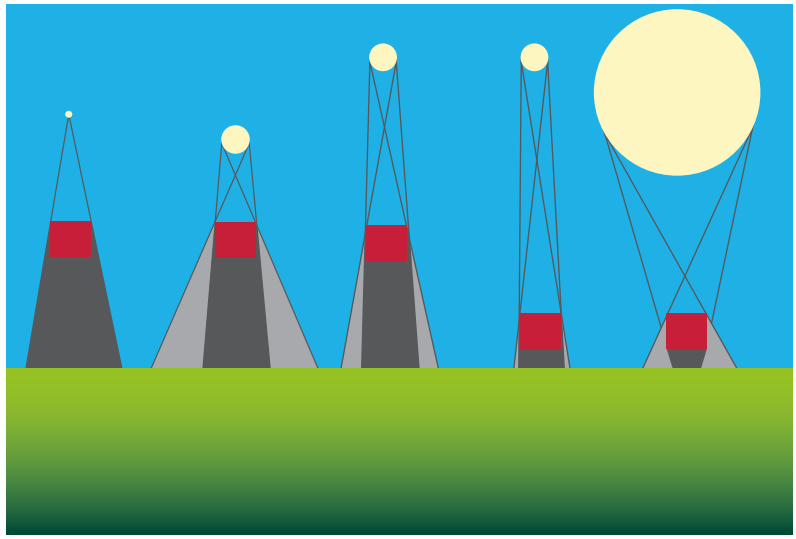


Figure 24-10. *Geometry of shadows. Dark gray is full shadow (umbra), and light gray is partial shadow (penumbra).*

multiplier for the light's contribution to shading. To do this, we will just shoot rays from the surface that we are shading to random points on the light source and calculate the percentage of rays that hit the light. We are also going to limit our examples to spherical positional lights, spherical spotlights, and circular directional lights, but the techniques apply to other light shapes and light types.

When we shoot multiple rays toward different places on a light source, we start to get noise at the edges of the shadow. We get noise in the penumbra, which is Latin for "almost a shadow" (Figure 24-9, right). The penumbra is the only place where there is noise because all other places are either completely in light or completely in shadow (umbra) and all rays will agree on visibility of the light. The size of the penumbra is based on the size and distance of the light source, as well as the size and distance of the shadow caster. Figure 24-10 shows the geometric relationships.

24.7.2 SPHERICAL DIRECTIONAL LIGHTS

The sun is a good example of a spherical directional light—a glowing ball that is so far away that no matter where you move, it will always be at the same point, angle, and direction in the sky.

We see the ball as a circle, though, so to do ray traced shadows, we are going to shoot rays at points on that circle and see what percentage were blocked for our shadowing term.

Doing this ends up being easy. We need to define the direction to the light and the solid angle radius of the circle in the sky, then we can use this GLSL code to get a direction in which to shoot a ray for a single visibility test:

```
// rect.x and rect.y are between 0 and 1.
vec2 MapRectToCircle(in vec2 rect)
{
  float radius = sqrt(rect.x);
  float angle = rect.y * 2.0 * 3.14159265359;
  return vec2(
    radius * cos(angle),
    radius * sin(angle)
  );
}

// rect.x and rect.y are between 0 and 1. direction is normalized direction
     to light. radius could be ~0.1.
vec3 SphericalDirectionalLightRayDirection(in vec2 rect, in vec3 direction,
     in float radius)
{
  vec2 point = MapRectToCircle(rect) * radius;
  vec3 tangent = normalize(cross(direction, vec3(0.0, 1.0, 0.0)));
  vec3 bitangent = normalize(cross(tangent, direction));
  return normalize(direction + point.x * tangent + point.y * bitangent);
}
```

To shoot eight shadow rays for a pixel, you would generate eight random vec2s, put them through `SphericalDirectionalLightRayDirection()` to get eight ray directions for that light, and multiply the percentage of hits by the lighting from this light. That gives you ray traced soft shadows from area lights.

As we know from previous sections though, not all random numbers are created equal!

The first thing we are going to do is use blue noise samples in a square, instead of white noise. If we use the same samples per pixel, we get the strange patterns you see in Figure 24-11.

The typical way to address all pixels using the same sampling pattern is to use Cranley–Patterson rotation: you use a per-pixel random number to modify the samples in some way to make them different. Instead of using regular random numbers, we can instead read a screen-space tiled blue noise mask texture as our source of per-pixel random numbers. We need two blue noise values, so we will read at (x, y) and will also read at an offset

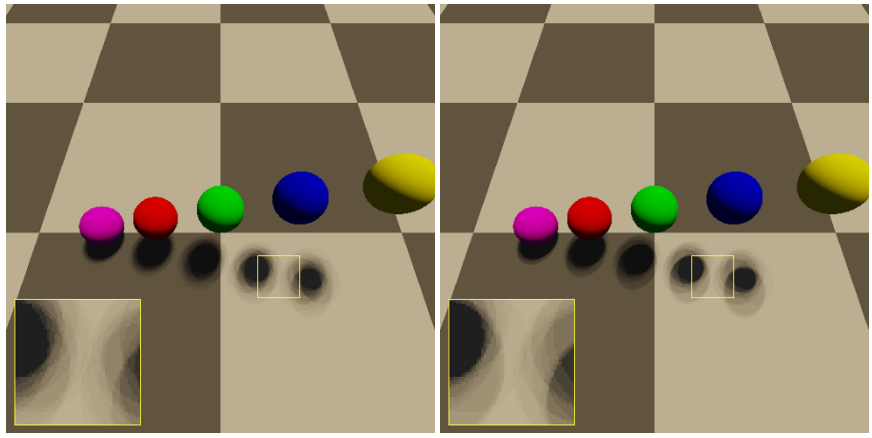


Figure 24-11. *The same 16 samples for each pixel. Left: blue noise. Right: white noise.*

$(x + 13, y + 41)$ to get two values between 0 and 1. The two values read from the texture are an (x, y) offset that we will add to each 2D blue noise sample, and we use `fract` to keep them between 0 and 1. When we do that, we get what we see in the leftmost column of Figure 24-12.

The offset of (13, 41) is somewhat arbitrary, but it was chosen to be not very close on the blue noise texture to where we read previously. Blue noise textures have correlation between pixels over small distances, so if you want uncorrelated values, you don't want to read too closely to the previous read location.

What we have looks good for still images, but in real-time graphics we also have the time dimension. For algorithms that temporally integrate rendering, such as temporal antialiasing (TAA) or deep learning super sampling (DLSS), using the same blue noise every frame is not giving them any new information, which leaves image quality unrealized. Even without temporal integration algorithms, better temporal sampling may look better to our eyes, or be implicitly integrated by the display. Individual screenshots will not look as nice if they aren't filtered over time, however.

The most straightforward way to animate these soft shadows would be to have something like eight different blue noise textures instead of one and to use the texture `frameNumber % 8` for a specific frame. This would animate the blue noise ray traced soft shadows, and every single frame would be good quality, but the problem is that looking at an individual pixel, it would become white noise over time. This is because each blue noise texture was made

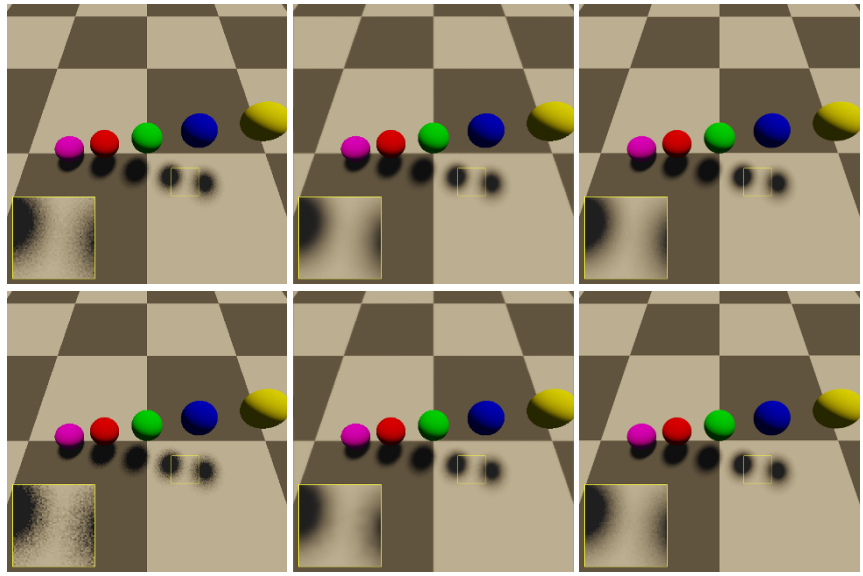


Figure 24-12. *Example of spherical directional light. Top: blue noise. Bottom: white noise. Left to right: raw, depth-aware Gaussian blur, and temporal antialiasing. Notice how the Gaussian blurred penumbra of the white noise looks much lumpier than the blue noise. This is because blue noise does not have frequencies low enough to survive the low pass filter, but white noise does.*

independently of the others. We know that white noise sampling is among the worst you can do usually, so we should be able to do better.

Another way would be to offset the texture reads by some amount every frame, essentially having a global texel offset for the pixels to add to their read location into the blue noise. This is what was done in INSIDE [8] using a Halton sequence to get the offset. Blue noise has correlation over small distances, but not over large distances, so doing this with a low discrepancy sequence results in white noise over time. This gives similar results to flipping through multiple textures, but has the benefit of only being a single texture, and you can have a flip cycle up to as long as the number of pixels in your texture.

Another way to animate blue noise is to read a blue noise texture, add the frame number multiplied by the golden ratio to it, then use modulus to keep it between 0 and 1. What this does is make each pixel use the golden ratio additive recurrence low-discrepancy sequence on the time axis, which is a high-quality 1D sampling sequence, making the time axis well-sampled. Unfortunately, this comes at the cost of modifying frequency content, so it damages the quality over space a little bit. Overall, it is a net win, though, and

this is the method that we are going to use. You can see this animated noise under TAA compared to white noise in the right most column of Figure 24-12.

```
float AnimateBlueNoise(in float blueNoise, in int frameIndex)
{
  return fract(blueNoise + float(frameIndex % 32) * 0.61803399);
}
```

You might think that the correct way to animate a 2D blue noise mask would be to make a 3D volumetric blue noise texture, but surprisingly, it is not, and 2D slices of 3D blue noise are not good blue noise at all [18]. You might also think that you could scroll a blue noise texture over time because neighboring pixels are blue noise in relation to each other, so that should make samples blue over time. This ends up not working well because 1D slices of 2D blue noise are also not good 1D blue noise, so you do not get good sampling over time; also, there will be a correlation between your samples along the scrolling direction.

Besides temporal integration and good noise patterns over time, you can also look at denoising a single frame in isolation. In the middle column of Figure 24-12, we use a depth-aware Gaussian blur on the lighting contributions before multiplying by albedo to get the final render.

At this point, we have spherical directional lights that have decent noise characteristics over both space and time.

24.7.3 SPHERICAL POSITIONAL LIGHTS

Spherical positional lights are glowing balls in the world. As you move around, the direction to them changes, and so does their brightness through distance attenuation. You can see one rendered in Figure 24-13.

Starting with the spherical directional light shader code, we only need to make two changes:

1. Light direction is no longer constant but is whatever the direction from the surface to the center of the light is.

2. The radius of the light circle is no longer constant either but changes as you get closer or farther from the light. Though being the correct thing to do, this also handles distance attenuation implicitly.

Here is the spherical directional light code modified to calculate the direction and radius as the first two lines, instead of taking them as parameters to the function. The rest works the same as before.

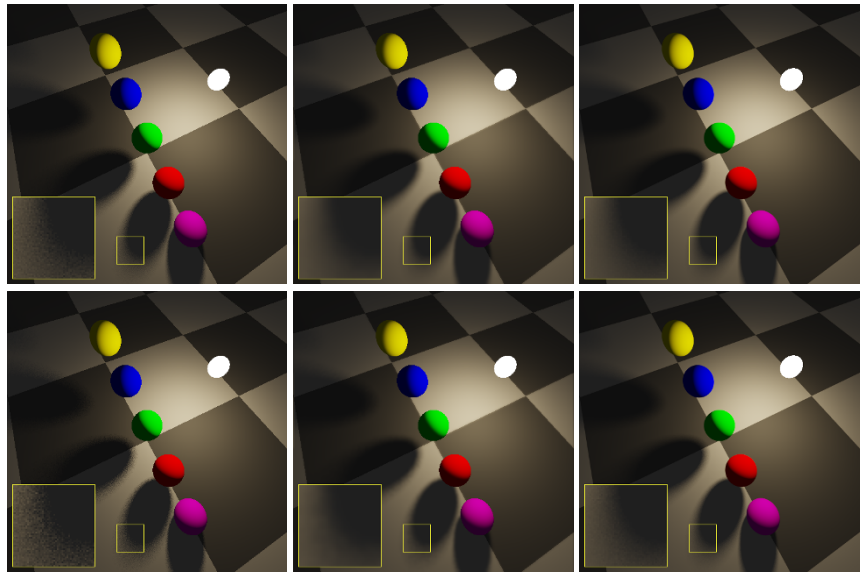


Figure 24-13. *Example of spherical positional light. Top: blue noise. Bottom: white noise. Left to right: raw, depth-aware Gaussian blur, and TAA.*

```
// rect.x and rect.y are between 0 and 1. surfacePos and lightPos are in
    world space.  worldRadius is in world units and could be ~5.
vec3 SphericalPositionalLightRayDirection(in vec2 rect, in vec3 surfacePos,
    in vec3 lightPos, in float worldRadius)
{
  vec3 direction = normalize(lightPos - surfacePos);
  float radius = worldRadius / length(lightPos - surfacePos);

  vec2 point = MapRectToCircle(rect) * radius;
  vec3 tangent = normalize(cross(direction, vec3(0.0, 1.0, 0.0)));
  vec3 bitangent = normalize(cross(tangent, direction));
  return normalize(direction + point.x * tangent + point.y * bitangent);
}
```

24.7.4 SPHERICAL SPOTLIGHTS

Last comes spherical spotlights, which are just like spherical positional lights except that they do not emit light from all angles. You can see one rendered in Figure 24-14.

To account for this, we are going to modify the spherical positional light code to take a spotlight direction as a parameter to control where the light is shining, and a cosine inner and cosine outer parameter to control how focused it is in that direction.

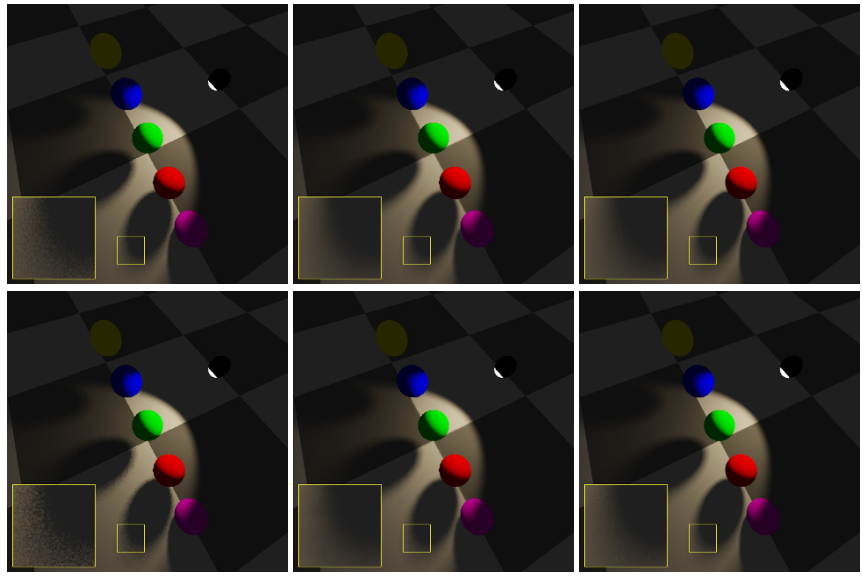


Figure 24-14. *Example of spherical spotlight. Top: blue noise. Bottom: white noise. Left to right: raw, depth-aware Gaussian blur, and TAA.*

We are going to dot product the direction from the light to the surface, by the direction in which the light shines. If it is less than the cosine inner value, it is fully bright. If it is greater than the cosine outer value, it is fully dark. If it is in between, we will use `smoothstep` (a cubic spline) to interpolate between fully bright and fully dark. This brightness value will be returned to the caller, which can be multiplied by the shadow sample result (0 or 1) before being averaged into the final result.

```
// rect.x and rect.y are between 0 and 1. surfacePos and lightPos are in
      world space.  worldRadius is in world units and could be ~5. angleAtten
       will be between 0 and 1.
vec3 SphericalSpotLightRayDirection(in vec2 rect, in vec3 surfacePos, in
      vec3 lightPos, in float worldRadius, in vec3 shineDir, in float
      cosThetaInner, in float cosThetaOuter, out float angleAtten)
{
  vec3 direction = normalize(lightPos - surfacePos);
  float radius = worldRadius / length(lightPos - surfacePos);

  angleAtten = dot(direction, -shineDir);
  angleAtten = smoothstep(cosThetaOuter, cosThetaInner, angleAtten);

  vec2 point = MapRectToCircle(rect) * radius;
  vec3 tangent = normalize(cross(direction, vec3(0.0, 1.0, 0.0)));
  vec3 bitangent = normalize(cross(tangent, direction));
  return normalize(direction + point.x * tangent + point.y * bitangent);
}
```

24.7.5 REDUCING RAY COUNT

One way to reduce the number of rays used for shadow rays is to shoot a few rays to start and, if they all agree on visibility, return the result as the answer. This makes an educated guess about the surface being either fully lit or fully shadowed. The more rays used, the more often this is correct, but the more expensive it is. If those few rays do not all agree, you know that you are in the penumbra and so should shoot more rays for better results.

You can also use a hybrid approach where you have a shadow map, but you only use it to tell if you are in the penumbra or not. You would do this by reading a shadow map value and, if you get a full 0 or 1 back as a result, use that without shooting any rays; otherwise, you would shoot your rays to get the penumbra shadowed value. Regular shadow maps are made from discrete light sources though, so for best results you probably would want to use something like percentage-closer soft shadows (PCSS) [6].

You can reduce the ray count all the way to one sample per pixel (spp) and still get somewhat decent results, as shown in Figure 24-15. You can take it below 1 spp by doing a ray trace for shadows smaller than full resolution, then filtering or interpolating the results for each individual pixel.

24.7.6 REDUCING NOISE

Noise is more apparent where there is higher contrast between the shadowed and unshadowed areas. In real life, shadows are almost never completely black, but instead have indirect global illumination bounce lighting of some amount in them. In this way, global illumination, ambient lighting, or similar can help reduce the appearance of noise without reducing the noise, but just reducing contrast.

Sample count also affects the contrast of your noise. If you only take one sample per pixel, you can only have 0 or 1 as a result, which means that the pixel is fully bright or fully in shadow, so your noise will all be either fully bright or fully in shadow. If you take two samples per pixel, you now have three values: fully in shadow, half in shadow, and fully in light. The noise in your shadows will then also have those three values and will have less contrast. The more samples you add, the more shades you have, with the shade count being the number of rays plus one.

Though we've been talking about shooting a ray toward a uniform random point on a light source, there are ways to sample area light sources that result

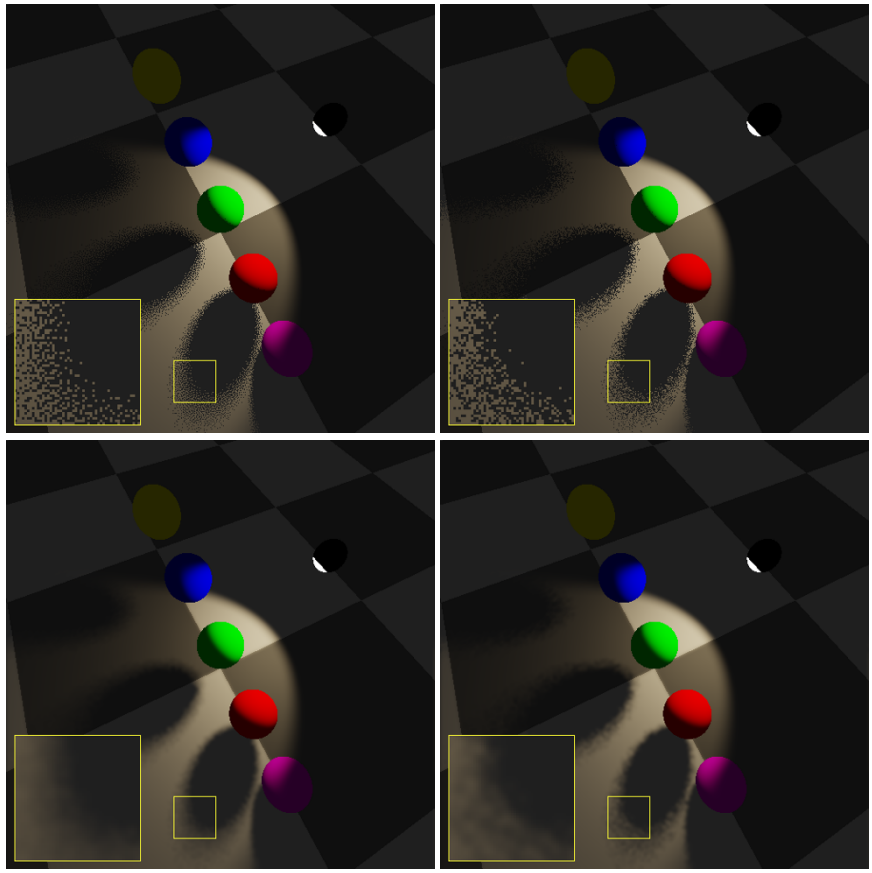


Figure 24-15. *One spp shadows. Top: raw. Bottom: depth-aware Gaussian blurred with $\sigma = 2$. Left: blue noise. Right: white noise.*

in less noise, to which you can then apply blue noise. You can read about some by Hart et al. [9] and Schutte [20].

When denoising ray traced shadows, it is best to keep your albedo, diffuse lighting, and specular lighting in separate buffers so that you can denoise your diffuse and specular lighting separately, before multiplying by albedo to get the final render. If you denoise the final render directly with a simpler denoising technique such as a depth-aware Gaussian blur, you will also end up blurring details in your albedo that come from textures, despite them not being noisy in the first place.

This chapter has focused on combining blue noise samples with blue noise masks, but you can use either of these things independently. For instance,

you might find that you get better convergence rates by using another sampling sequence per pixel, but still using a blue noise mask to vary the sampling sequence per pixel, making the resulting error pattern be blue noise in screen space.

Noise appears only in penumbras, and smaller light sources make smaller penumbras, so you can try shrinking your lights if you have too much noise. Glancing angles of lighting also make for longer penumbras, and larger areas of noise, like the sun being low in the sky, making long shadows. If you can shrink your penumbra, you will shrink the area where noise can be.

24.8 COMPARISON WITH INTERLEAVED GRADIENT NOISE

Interleaved gradient noise was presented by Jimenez [21] and is tailored toward temporal antialiasing. IGN is fast to generate from an x and y pixel coordinate, so there is no precomputed texture or a texture read. In this way, IGN is a competitor to blue noise in that it is used for the same things, is fast, and gives situationally desirable properties.

```
float IGN(int x, int y)  // x and y are in pixels.
{
  return fract(52.9829189f * fract(0.06711056f*float(x)
                                   + 0.00583715f*float(y)));
}
```

In most modern TAA implementations, a pixel will sample its 3×3 neighborhood to get a minimum and maximum color cube, possibly in another color space such as YCoCg. The previous frame's pixel color is then constrained to this color cube before interpolating toward the current frame's pixel color, possibly using a reversible tone mapping operation [23]. This constraint is used to effectively keep or reject history based on how different the history is from the local neighborhood of this frame.

If you wanted to do something like stochastic transparency for an object that has 1/9th transparency, during the G-buffer fill, you could use a per-pixel random value (white noise) and discard the pixel write if that random value was greater than the transparency value. This makes 1/9 of the pixels survive.

For the neighborhood color constraint, what you would hope for is that out of every 3×3 group of pixels under stochastic alpha, exactly one of them should survive the alpha test to most accurately represent the color cube. The problem is that white noise has clumps and voids, so there will be many 3×3 groups of pixels that have no surviving pixels, and others that have more than

one. When there are no pixels, the history is erroneously rejected and the semitransparent pixel does not contribute to the final pixel color. When there is more than one pixel, that color is over-represented, and it will look more opaque than it is.

If you use blue noise for this instead of white noise, the pixels that survive the alpha test will be roughly uniform; So, often one pixel will survive in every 3×3 block of pixels, but this is not guaranteed. Sometimes you will have more or less.

Interleaved gradient noise does guarantee this, however. This is because every 3×3 block of pixels in IGN has all values approximately 0/9, 1/9, 2/9, . . . 8/9. This is a generalized Sudoku that is impossible to solve because there are too many constraints. IGN gets around this by having small numerical drift, which also makes IGN values fully continuous, unlike blue noise, which often comes from a U8 texture and so only has 256 different possible values.

The same situation seen in stochastic transparency comes up when you ray-trace shadows (or doing other techniques). Using IGN as a per-pixel random value means that in every 3×3 group of pixels, you'll have a histogram of results that more closely matches the actual possible histogram of results, compared to white noise. That leads to more accurate history acceptance and rejection.

So, if you are using TAA with neighborhood sampling for history rejection, you may want to try using IGN instead of a blue noise mask, as a per-pixel random number. Figure 24-16 shows a comparison.

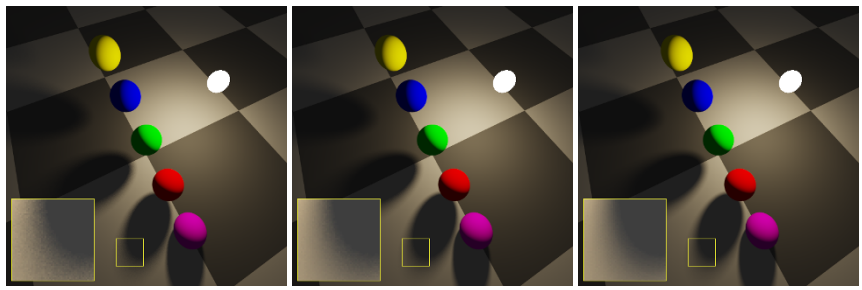


Figure 24-16. *Sixteen spp shadows under TAA. Inset images have +1 fstop exposure to make differences more visible. Left to right: white noise, blue noise samples with blue noise mask, and blue noise samples with IGN mask.*

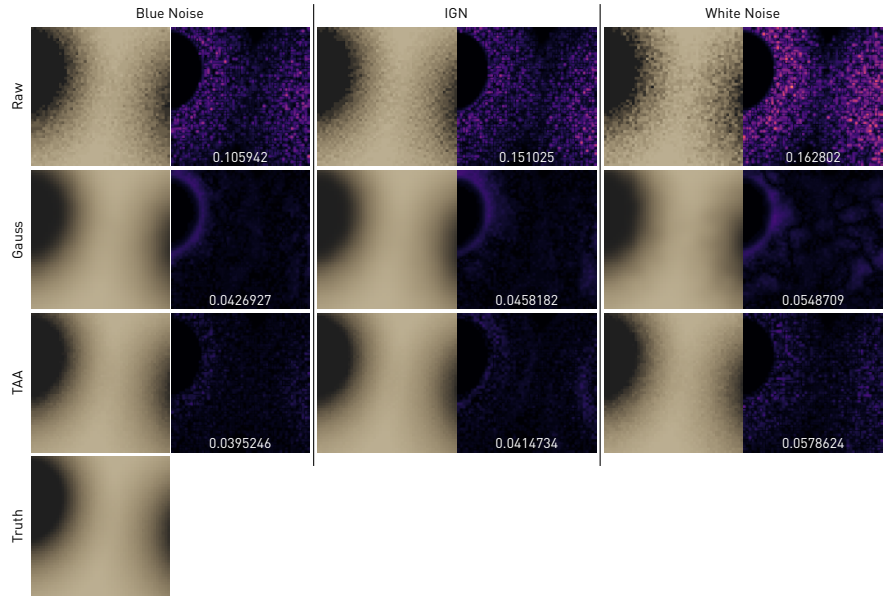


Figure 24-17. *Directional light scene ꟻLIP means (inset number) and heat maps (right images). Truth is ten thousand white noise samples.*

24.9 PERCEPTUAL ERROR EVALUATION

As we've seen, different arrangements of error can affect how an image looks even if it has the same amount of error. Because of this, perceptual error metrics are an active area of research.

Results of comparisons using the ꟻLIP perceptual error metric [3] are shown in Figure 24-17. The heat map of perceptual error is shown, as well as the mean. Lower mean values are better.

ꟻLIP tells us that blue noise is best in all situations, IGN is a little bit worse, and white noise is significantly worse. It's a little surprising that IGN didn't get a better score than blue noise under TAA since it seems to have less noisy pixels, but the scores are very close. Similarly, TAA in general seems to get a better score than Gaussian blurred results, despite looking a little noisier.

See also Chapter 19 about the ꟻLIP metric.

24.10 CONCLUSION

Offline rendering has a wealth of knowledge to give real-time rendering in the way of importance sampling and low-discrepancy sequences, but real-time rendering has much lower rendering computation budgets.

Because of this, we need to think more about what to do with four samples per pixel, one sample per pixel, or fewer than one sample per pixel, which is not as big of a topic in offline rendering.

Sampling well over both space and time are important, as are considerations to the frequency content of the noise for filtering and perceptual qualities. Blue noise can be a great choice, but it is not the only tool for the job as we saw with IGN's effectiveness under TAA. If you ever find yourself using white noise, though, chances are that you are leaving money on the table.

I believe that in the future this topic will continue to bloom, allowing us to squeeze every last drop of image quality out of our rendering capabilities, with the same number of rays.

REFERENCES

[1] Ahmed, A. G. M., Perrier, H., Coeurjolly, D., Ostromoukhov, V., Guo, J., Yan, D.-M., Huang, H., and Deussen, O. Low-discrepancy blue noise sampling. *ACM Transactions on Graphics*, 35(6):247:1–247:13, Nov. 2016. DOI: 10.1145/2980179.2980218.

[2] Ahmed, A. G. M. and Wonka, P. Screen-space blue-noise diffusion of Monte Carlo sampling error via hierarchical ordering of pixels. *ACM Transactions on Graphics*, 39(6):244:1–244:15, Nov. 2020. DOI: 10.1145/3414685.3417881.

[3] Andersson, P., Nilsson, J., Akenine-Möller, T., Oskarsson, M., Åström, K., and Fairchild, M. D. ꟻLIP: A difference evaluator for alternating images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2):15:1–15:23, 2020. DOI: 10.1145/3406183.

[4] Christensen, P., Kensler, A., and Kilpatrick, C. Progressive multi-jittered sample sequences. *Computer Graphics Forum*, 37(4):21–33, 2018. DOI: 10.1111/cgf.13472.

[5] De Goes, F., Breeden, K., Ostromoukhov, V., and Desbrun, M. Blue noise through optimal transport. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 31:171:1–171:11, 6, 2012. DOI: 10.1145/2366145.2366190.

[6] Fernando, R. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, 35–es, 2005. DOI: 10.1145/1187112.1187153.

[7] Georgiev, I. and Fajardo, M. Blue-noise dithered sampling. In *ACM SIGGRAPH 2016 Talks*, 35:1, 2016. DOI: 10.1145/2897839.2927430.

[8] Gjoel, M. and Svendsen, M. The rendering of INSIDE: High fidelity, low complexity. Game Developer's Conference, https://www.gdcvault.com/play/1023002/Low-Complexity-High-Fidelity-INSIDE, 2016.

[9] Hart, D., Pharr, M., Müller, T., Lopes, W., McGuire, M., and Shirley, P. Practical product sampling by fitting and composing warps. Eurographics Symposium on Rendering (EGSR'20), http://casual-effects.com/research/Hart2020Sampling/index.html, 2020.

[10] Heitz, E., Belcour, L., Ostromoukhov, V., Coeurjolly, D., and Iehl, J.-C. A low-discrepancy sampler that distributes Monte Carlo errors as a blue noise in screen space. In *ACM SIGGRAPH 2019 Talks*, 68:1–68:2, 2019. DOI: 10.1145/3306307.3328191.

[11] Heitz, E., Hill, S., and McGuire, M. Combining analytic direct illumination and stochastic shadows. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2:1–2:11, 2018. DOI: 10.1145/3190834.3190852.

[12] Jarosz, W., Enayet, A., Kensler, A., Kilpatrick, C., and Christensen, P. Orthogonal array sampling for Monte Carlo rendering. *Computer Graphics Forum*, 38(4):135–147, 2019. DOI: 10.1111/cgf.13777.

[13] Keller, A., Georgiev, I., Ahmed, A., Christensen, P., and Pharr, M. My favorite samples. In *ACM SIGGRAPH 2019 Courses*, 15:1–15:271, 2019. DOI: 10.1145/3305366.3329901.

[14] Kopf, J., Cohen-Or, D., Deussen, O., and Lischinski, D. Recursive Wang tiles for real-time blue noise. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)*, 25(3):509–518, 2006. DOI: 10.1145/1179352.1141916.

[15] Mitchell, D. P. Spectrally optimal sampling for distribution ray tracing. *SIGGRAPH Computer Graphics*, 25(4):157–164, July 1991. DOI: 10.1145/127719.122736.

[16] Perrier, H., Coeurjolly, D., Xie, F., Pharr, M., Hanrahan, P., and Ostromoukhov, V. Sequences with low-discrepancy blue-noise 2-D projections. *Computer Graphics Forum (Proceedings of Eurographics)*, 37(2):339–353, 2018. https://hal.archives-ouvertes.fr/hal-01717945.

[17] Peters, C. Free blue noise textures. http://momentsingraphics.de/BlueNoise.html, 2016. Accessed January 15, 2021.

[18] Peters, C. The problem with 3D blue noise. http://momentsingraphics.de/3DBlueNoise.html, 2017. Accessed February 8, 2021.

[19] Reinert, B., Ritschel, T., Seidel, H.-P., and Georgiev, I. Projective blue-noise sampling. *Computer Graphics Forum*, 35(1):285–295, 2015. DOI: 10.1111/cgf.12725.

[20] Schutte, J. Sampling the solid angle of area light sources. https://schuttejoe.github.io/post/arealightsampling/, 2018. Accessed January 20, 2021.

[21] Tatarchuk, N., Karis, B., Drobot, M., Schulz, N., Charles, J., and Mader, T. Advances in real-time rendering in games, Part I. In *ACM SIGGRAPH 2014 Courses*, 10:1, 2014. DOI: 10.1145/2614028.2615455.

[22] Ulichney, R. A. Void-and-cluster method for dither array generation. In J. P. Allebach and B. E. Rogowitz, editors, *Human Vision, Visual Processing, and Digital Display IV*, volume 1913 of *Proceedings*, pages 332–343. SPIE, 1993. DOI: 10.1117/12.152707.

[23] Yang, L., Liu, S., and Salvi, M. A survey of temporal antialiasing techniques. *Computer Graphics Forum*, 39(2):607–621, 2020. DOI: https://doi.org/10.1111/cgf.14018.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



PART IV
SHADING AND EFFECTS

PART IV

SHADING AND EFFECTS

Ray tracing excels with its ability to aid shading and various rendering effects. Ray traversal can be used in a traditional way to explore light paths, or it can be employed as a general-purpose tool for querying scene information. The techniques presented in this part cover a wide range of topics related to efficiently gathering information needed for shading, primary/secondary effects, and adaptations of common rasterization rendering methods for a ray tracing framework.

Temporal filtering is an essential tool to minimize the number of rays needed for properly resolving secondary effects with high performance on today's GPUs. In dynamic scenes, this requires motion vectors for accessing the relevant information at the previous frame. Chapter 25, *Temporally Reliable Motion Vectors for Better Use of Temporal Information*, presents specialized motion vectors for moving shadows, glossy reflections, and occlusions.

Chapter 26, *Ray Traced Level of Detail Cross-Fades Made Easy*, describes simple and efficient methods for supporting level of detail techniques that are commonplace with rasterization. This is particularly important for hybrid renderers that combine ray tracing and rasterization.

Chapter 27, *Ray Tracing Decals*, explains how decals, which are commonly used in 3D games, can be incorporated into ray tracing, so that they can appear in secondary effects computed using ray tracing, such as reflections.

Using impostors to represent complex objects that are expensive to render is another common technique with rasterization-based rendering. Chapter 28, *Billboard Ray Tracing for Impostors and Volumetric Effects*, shows how impostors can be handled with ray tracing for replacing objects with high geometric complexity or volumetric media.

Correctly handling refractions requires ray tracing, though image-space refractions can provide a fast approximation of up to two refraction events. Chapter 29, *Hybrid Ray Traced and Image-Space Refractions*, describes different ways of combining these two approaches to reduce the number of rays needed to account for more than two refraction events, which is important when rendering typical refractive models such as water and glass.

Highly specular objects focus reflected and refracted light, forming caustics, which can be expensive to compute. Chapter 30, *Real-Time Ray Traced Caustics*, provides an efficient photon-based solution for rendering caustics that uses temporal information to improve the sample density for the next frame in a real-time renderer.

One of the advantages of ray tracing over rasterization is its ability to handle complex camera models. Chapter 31, *Tilt-Shift Rendering Using a Thin Lens Model*, describes methods for simulating a camera with a tilt-shift lens for achieving views with the desired perspective distortion or an arbitrary focal plane that is not necessarily aligned with the view direction.

Cem Yuksel

CHAPTER 25

TEMPORALLY RELIABLE MOTION VECTORS FOR BETTER USE OF TEMPORAL INFORMATION

Zheng Zeng,[1] Shiqiu Liu,[2] Jinglei Yang,[3] Lu Wang,[1] and Ling-Qi Yan[3]
[1]Shandong University
[2]NVIDIA
[3]University of California, Santa Barbara

ABSTRACT

We present temporally reliable motion vectors that aim at deeper exploration of temporal coherence, especially for the generally believed difficult applications on shadows, glossy reflections, and occlusions. We show that our temporally reliable motion vectors produce significantly more robust temporal results than current traditional motion vectors while introducing negligible overhead.

25.1 INTRODUCTION

The state-of-the-art reconstruction methods [2, 9, 7, 10] all rely on temporal filtering. Though it is demonstrated to be powerful, robust temporal reuse has been very challenging due to the fact that motion vectors are sometimes not valid. For example, a static location in the background may be blocked by a moving object in the previous frame. In this case, the motion vector does not exist at the current location. Also, the motion vectors may be wrong for effects like shadows and reflections. For example, a static shadow receiver will always have a zero-length motion vector, but the shadows casted onto it may move arbitrarily along with the light source. In any of these cases, when correct motion vectors are not available but temporal filtering is applied anyway, ghosting artifacts (unreasonable leak or lag of shading over time) will emerge.

Although the temporal failures can be detected with smart heuristics [10], the temporal information in these cases will be *simply rejected* nonetheless. But we believe that the information can be *better utilized*. In this chapter, we

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_25

present different types of motion vectors for different effects, to make the seemingly unusable temporal information available again. Specifically, we introduce the following:

- A shadow motion vector for moving shadows.
- A stochastic glossy reflection motion vector for glossy reflections.
- A dual motion vector for occlusions.

25.2 BACKGROUND

In this section, we briefly go over the calculation of traditional motion vectors, and explain how they are used in temporal filtering.

When two consecutive frames $i-1$ (previous) and i (current) are given, the idea of back-projection is to find for each pixel X_i intersected by the primary ray, where its world-space shading point S_i was in the previous frame at X_{i-1}. Because we know the entire rendering process, the back-projection process can be accurately computed: first, project the pixel X_i back to its world coordinate in the ith frame, then transform it back to the $(i-1)$-th frame according to the movement of the geometry, and finally project the transformed world coordinate in the $(i-1)$-th frame back to the image space to get X_{i-1}. Denote $\mathbf{P} = \mathbf{M}_v \mathbf{M}_{mvp}$ as the viewport $\mathbf{M}_v$ times the model-view-projection transformation $\mathbf{M}_{mvp}$ per frame and $\mathbf{T}$ as the geometry transformation between frames; then, the back-projection process can be formally written as

$$X_{i-1} = \mathbf{P}_{i-1}\mathbf{T}^{-1}\mathbf{P}_i^{-1}X_i, \tag{25.1}$$

where the subscripts represent different frames. According to Equation 25.1, the motion vector $\mathbf{m}(X_i) = X_{i-1} - X_i$ is defined as the difference between the back-projected pixel and the current pixel in the image space. The following pseudocode shows how this traditional motion vector is calculated:

```
calcTradMotionVector(uint2 pixelIndex, float4 hitPos, uint hitMeshID){
    // For each pixel X_i (in image space) ...
    float2 X = pixelIndex + float2(0.5f, 0.5f);
    // ... find its world-space shading point S_i.
    float4 S = hitPos;
    // Then, transform S_i back to the previous frame to get S_(i-1).
    // The custom function getT(...) returns the geometry transformation for
         a given mesh, and inverse(...) inverts a 4x4 matrix.
    float4x4 invT = inverse(getT(hitMeshID));
    float4 prevS = mul(S, invT)
    // Finally, project it to screen space to get X_(i-1).
    // The custom function toScreen(...) projects a given world-space point
        to screen space to get corresponding the image-space pixel.
```

```
12      float2 prevX = toScreen(prevS);
13      // Return traditional motion vector.
14      return prevX - X;
15  }
```

The motion vector for each pixel X_i is computed together with the rendering process and can be acquired almost without any performance overhead. With the motion vectors, temporal filtering becomes straightforward. In practice, it is a simple linear blending between the current and previous pixel values:

$$\bar{c}_i(X_i) = \alpha \cdot \tilde{c}_i(X_i) + (1 - \alpha) \cdot \bar{c}_{i-1}(X_{i-1}), \tag{25.2}$$

where c is the pixel value, first filtered spatially per frame resulting in $\tilde{c}$, then blended with the pixel value of its previous correspondence $\bar{c}_{i-1}$. The α is a factor between 0 and 1 that determines how much temporal information is trusted and used, usually set to 0.1 to 0.2 in practice, indicating heavy temporal dependence on previous frames. The temporal filtering process continues as more and more frames are rendered, accumulating to a cleaner result. Thus, we use the $\bar{}$ and $\tilde{}$ symbols to indicate less and more noise, respectively.

From Equation 25.2, we can see that one frame's contribution over time is an exponential falloff. Thus, if the motion vectors cannot accurately represent correspondence between adjacent frames, ghosting artifacts will appear. Various methods are designed to alleviate the temporal failure. Salvi [8] proposed to clamp the previous pixel value $\bar{c}_{i-1}(X_{i-1})$ to the neighborhood of the current pixel value, in order to suppress the ghosting artifacts and provide a faster rate of convergence to the current frame. The spatiotemporal variance-guided filtering (SVGF) method [9] focuses on a better spatial filtering scheme to acquire $\tilde{c}_i(X_i)$ by considering spatial and temporal variances together. And the adaptive SVGF (A-SVGF) method [10] detects rapid temporal changes to adjust the blending factor α to rely more or less on spatial filtering, trading ghosting artifacts for noise.

25.3 TEMPORALLY RELIABLE MOTION VECTORS

In this section, we describe our temporally reliable motion vectors. Specifically, we will focus on the three commonly encountered temporal failure cases: shadows, glossy reflections, and occlusions. In contrast to the previous methods, we intend to better utilize the previous information, i.e., we would like to find a more reliable $\bar{c}_{i-1}(X_{i-1})$ so that minimal special treatment is further needed. Our insight is that for shadows and glossy reflections, it is

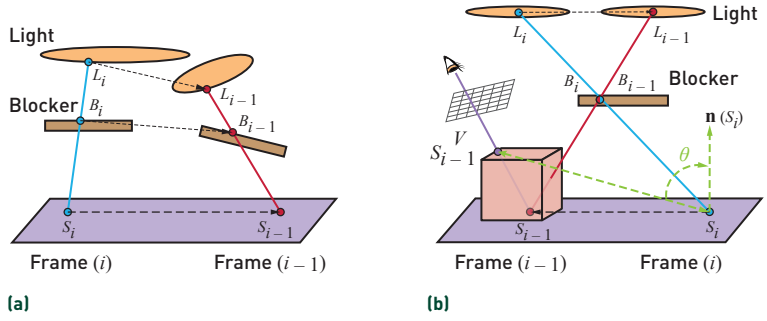


Figure 25-1. *(a) Computation of our shadow motion vectors. (b) The nonplanar shadow receiver issue.*

not the geometry in a pixel that we want to track in the previous frame, but the position of the shadow and the reflected virtual image. For occlusions, it is easier for the previously occluded regions in the background to find correspondences also in the background rather than on the occluder. After introducing the corresponding motion vectors, we then compare them with the state-of-the-art methods and report the computation cost.

25.3.1 SHADOWS

Inspired by percentage closer soft shadows (PCSS) [4], which estimate the shadow size based on the average blocker depth and light size, we propose to track the movement of shadows by following the blocker and light positions over time.

Figure 25-1a illustrates our scheme focusing on two consecutive frames (i – 1) and i. We shoot one shadow ray per pixel toward a randomly chosen position on the light. For a pixel X_i (in image space) in shadow, we know exactly its shading point S_i, the blocker position B_i, and the light sample position L_i (all in world space). Because the blocker and the light sample positions are associated with certain objects, we immediately know their transformation matrices between these two frames, so we are able to find their world-space positions B_{i-1} and L_{i-1}, respectively, in the (i – 1)-th frame. If the geometry around the shadow receiver S_i is a locally flat plane, we can find the intersection S_{i-1} between this plane and the line connecting L_{i-1} and B_{i-1} in the previous frame. Finally, we project this intersection to the screen space. In this way, this projected pixel X^V_{i-1} is our tracked shadow position from X_i:

$$X^V_{i-1} = \mathbf{P}_{i-1}\ \text{intersect}[\mathbf{T}^{-1}L_i \rightarrow \mathbf{T}^{-1}B_i; \mathbf{T}^{-1}\text{plane}(S_i)], \tag{25.3}$$

where the transformations **T** between frames can be different for the light sample, blocker, and shading point.

Equation 25.3 implies that our shadow motion vector is $\mathbf{m}^V(X_i) = X^V_{i-1} - X_i$. The pseudocode for calculating the shadow motion vector looks as follows:

```
calcShadowMotionVector(uint2 pixelIndex, float4 hitPos, float4 hitNormal,
     uint hitMeshID, float4 blockerPos, uint blockerMeshID, float4 lightPos,
      uint lightMeshID){
    // For each pixel X_i in shadow (in image space) ...
    float2 X = pixelIndex + float2(0.5f, 0.5f);
    // ... find its shading point S_i, blocker point B_i, and light sample
         point L_i (all in world space).
    float4 S = hitPos;
    float4 B = blockerPos;
    float4 L = lightPos;
    // Then, transform B_(i-1) and L(i-1) back to the previous frame.
    // The custom function getT(...) returns the geometry transformation for
          a given mesh, and inverse(...) inverts a 4x4 matrix.
    float4 prevB = mul(B, inverse(getT(blockerMeshID)));
    float4 prevL = mul(L, inverse(getT(lightMeshID)));
    // Next, find the intersection of the virtual plane (defined by S_i and
         its normal in the previous frame) and the ray (from L_(i-1) to B_(i
         -1)).
    float4 origin = prevL;
    float4 direction = prevB - prevL;
    float4x4 invT = inverse(getT(hitMeshID));
    float4 prevNormal = mul(hitNormal, invT);
    float4 prevS = mul(S, invT);
    // The custom function rayPlaneIntersect(...) finds the intersection
         between a ray (defined by origin and direction) and a plane (
         defined by point and normal).
    float4 intersection = rayPlaneIntersect(origin, direction, prevS,
         prevNormal);
    // Finally, project it to screen space to find X^(V)_(i-1).
    // The custom function toScreen(...) projects a given world-space point
         to screen space to get corresponding the image-space pixel.
    float2 prevX = toScreen(intersection);
    // Return the shadow motion vector.
    return prevX - X;
}
```

To use our shadow motion vectors, we slightly modify the temporal filtering Equation 25.2 by adding a lightweight clean-up filtering pass (at most 9×9) after temporal blending. This is because the motion vectors $\mathbf{m}^V(X_i)$ can be noisy due to random sampling on the light, so the fetched $\bar{V}_{i-1}(X_{i-1})$ can be noisy as well, despite the smoothness of $\bar{V}_{i-1}$ itself in the previous frame. The same clean-up filter will be used for glossy reflections and occlusions.

To perform spatial filtering of the noisy shadows in the current frame i, we refer to Liu et al. [6], which accurately calculates the filter size. However, we notice that based on the previous computation, only those pixels in shadows in

the current frame are associated with our shadow motion vectors. To deal with this problem, and to achieve both efficient filtering performance and clean shadow boundaries, we conceptually interpret the filtering of shadows as the splatting of each in-shadow pixel's visibility, along with other associated properties.

DISCUSSION: NONPLANAR SHADOW RECEIVER The only assumption we make is that the geometry is locally flat for each shading point during the computation of our shadow motion vectors. However, as Figure 25-1b shows, after the back-projection in the $(i-1)$-th frame, it is possible that S_{i-1}^V, the shading point of pixel X_{i-1}^V, is not on the virtual receiver plane defined by S_i and its normal $\mathbf{n}(S_i)$ (inverse transformed if the shadow receiver moves over time, omitted here for simplicity). These two shading points may not have the same normals and could even be on different objects. In this case, it seems that our shadow motion vector could no longer be used.

To deal with the problem introduced because of nonplanar shadow receivers, we introduce a simple but effective falloff heuristic. That is, we measure the extent of the "nonplanarity." Figure 25-1b illustrates our idea. Once X_{i-1}^V is calculated, we measure the angle θ between the normal of the virtual receiver plane $\mathbf{n}(S_i)$ and the direction $S_i \rightarrow S_{i-1}^V$. Our key observation is that, only when θ is close to 90°, we can fully depend on our shadow motion vector. Otherwise, we should trust more on the spatially filtered result. So, we use θ to adjust the α, replacing it with a specific α^V in Equation 25.2 as

$$\alpha^V = 1 - G\left(\theta - \frac{\pi}{2}; 0, 0.1\right) \cdot (1 - \alpha), \tag{25.4}$$

where $G(x; \mu, \sigma)$ is a Gaussian function with its peak value normalized to 1, centered at μ, and with a standard deviation of σ. Finally, we achieve high-quality, non-lagging shadows. This process looks as follows:

```
// Once X^(V)_(i-1) is calculated ...
float2 prevX = X + shadowMotionVector;
// ... measure the angle theta between the normal of S_i (in the previous
    frame) and the direction S_i->S^V_(i-1).
float4 S = hitPos;
// The custom function getPrevHitPos(...) returns the world-space shading
    point for a given pixel in the previous frame.
float4 prevSV = getPrevHitPos(prevX);
float3 direction = normalize(float3(prevSV - S));
// The custom function getT(...) returns the geometry transformation for a
    given mesh, and inverse(...) inverts a 4x4 matrix.
float3 prevNormal = float3(hitNormal, inverse(getT(hitMeshID)));
float theta = acosf(dot(direction, prevNormal))
// Use theta to adjust the alpha^V.
```

```
12 // The custom function gaussian(...) returns a value between 0 to 1
       according to theta.
13 float alphaV = 1.0f - gaussian(theta - PI / 2, 0.f, 0.1f) * (1.0f - alpha);
```

We compare our results with the ones generated using traditional motion vectors, with and without the neighborhood clamping approach used in temporal antialiasing (TAA) [8]. The clamping methods represent the line of ideas that force the use of temporal information. We also compare our method with the SVGF and A-SVGF methods as representatives that balance the use of temporal and spatial information. Neither kind of these methods aims at better utilizing the temporal information.

Figure 25-2 shows the *fence* scene with a rapidly moving fence in front and an area light behind it. In this example, we demonstrate that our shadow motion vectors are able to produce shadows that are closely attached to the fence.

In comparison, traditional motion vectors produce significant ghosting artifacts. This is expected because they will always be zero in this case. With clamping, the results are less lagging but much more noisy. However, the noise is aggressively filtered by SVGF, resulting in overblur. The A-SVGF method discards temporal information, resulting in color blocks similar to the typical "smearing" artifact in bilateral image filtering and leaving behind low-frequency noise that can be easily observed in a video sequence.

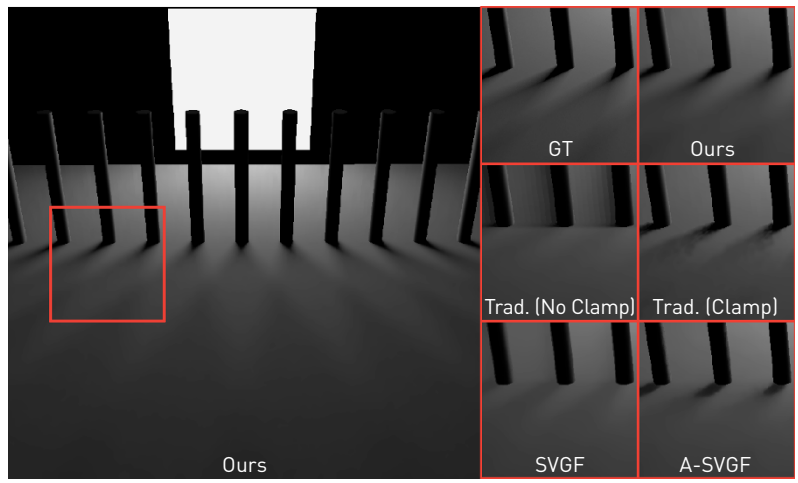


Figure 25-2. *The fence scene with a rapidly moving fence in front and an area light behind it. Our shadow motion vectors are able to produce shadows that are closely attached to the fence.*

25.3.2 GLOSSY REFLECTIONS

Similar to tracking the shadows, we can also track the movement of glossy reflections.

Our insight is that no matter whether we are using the specular or sampled direction, what we need to do is still find the corresponding pixel in the previous frame of the secondary hit point H_i. However, because glossy BRDFs model a non-delta distribution, multiple pixels may reflect to the same hit point, forming a finite area in the screen space. This indicates that there will be multiple pixels from the previous frame that correspond to X_i in the current frame. Our stochastic glossy motion vector aims at finding one at a time.

Given that the center of a glossy BRDF lobe is usually the strongest, we always have a valid choice with which to start. That is, we first assume that the glossy BRDF degenerates to pure specular, then we can immediately find the corresponding point S^C_{i-1} similar to Zimmer et al. [12]. Then, our insight is that, as the glossy lobe gradually emerges, a region will appear around S^C_{i-1} in which all the points are able to reflect to the same hit point H_{i-1}. This region can be approximated by tracing a glossy lobe (with the same roughness at S^C_{i-1}) from the virtual image of H_{i-1} toward S^C_{i-1}.

Figure 25-3 illustrates the way that we find one corresponding pixel X^R_{i-1} in the previous frame. We start from the importance-sampled secondary ray at the shading point S_i and the secondary hit point H_i in the world space. We transform H_i to the previous frame $(i - 1)$ in the world space, find its mirror-reflected image, then project it to the screen space to retrieve S^C_{i-1} in the world space again.

Then, similar to the shadow case, we assume a locally flat virtual plane around S^C_{i-1} and find the intersected region between this plane and the glossy lobe traced from the image of H_{i-1} toward S^C_{i-1}. In practice, there is no need to trace any cones, and we simply assume that the glossy lobe is a Gaussian in directions and that the intersected region is a Gaussian in positions as well as in the image space, which can be efficiently approximated by tracking the endpoints of major and minor axes. In this region, our stochastic motion vector for glossy reflection randomly finds X_i's correspondence at

$$X^R_{i-1} = \text{sample}\left(\mathbf{P}_{i-1}\ \text{mirror}[\mathbf{T}^{-1}H_i, \mathbf{T}^{-1}\text{plane}(S_i)], \Sigma\right), \tag{25.5}$$

where sample(μ, Σ) importance-samples a Gaussian function with center μ and covariance Σ, and $\mathbf{T}$ still represents different transformations at different

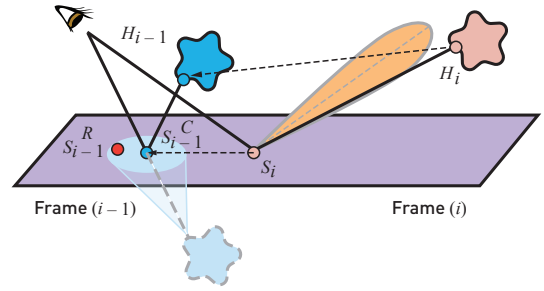


Figure 25-3. *The computation of our stochastic glossy reflection motion vectors. For an importance-sampled secondary ray, we back-project the virtual image of its hit point in the previous frame.*

places. The pseudocode for calculating our stochastic glossy motion vector looks as follows:

```
calcGlossyMotionVector(uint2 pixelIndex, float4 hitPos, float4 hitNormal,
    uint hitMeshID, float4 secondaryHitPos, uint secondaryHitMeshID){
    // For each pixel X_i, find the secondary hit point H_i.
    float2 X = pixelIndex + float2(0.5f, 0.5f);
    float4 H = secondaryHitPos;
    // Transform H_i back to the previous frame.
    // The custom function getT(...) returns the geometry transformation for
        a given mesh, and inverse(...) inverts a 4x4 matrix.
    float4 prevH = mul(H, inverse(getT(secondaryHitMeshID)));
    // Then, find its mirror-reflected image using the plane defined by S_i
        and its normal in the previous frame.
    float4 S = hitPos;
    float4x4 invT = inverse(getT(hitMeshID));
    float4 prevNormal = mul(hitNormal, invT);
    float4 prevS = mul(S, invT);
    // The custom function mirror(...) finds the mirror-reflected image
        behind a given plane for a given point.
    float4 mirroredPrevH = mirror(prevH, prevS, prevNormal);
    // Next, project it to screen space to find the "center."
    // The custom function toScreen(...) projects a given world-space point
        to screen space to get the corresponding image-space pixel.
    float2 center = toScreen(mirroredPrevH);
    // Finally, sample one position in image space around this center.
    // The key idea here is to simply approximate a 2D anisotropic Gaussian
        on the plane, then project it to screen space.
    // Assume that the glossy lobe is a Gaussian in directions with variance
        sigma (using custom function GaussianDist(...)); then, we can
        immediately find a 2D Gaussian on the plane according to the
        distance from the lobe to the plane (using custom function toPlane
        (...)).
    float2x2 covariance = toPlane(gaussianDist(sigma), prevS, prevNormal)
    // Then, project it to screen space to get the covariance of this 2D
        anisotropic Gaussian on screen.
    covariance = toScreen(covariance);
```

```
    // The custom function sampleGaussian2D(...) samples a 2D point in image
         space according to the Gaussian function with given center and
        covariance.
    float2 prevX = sampleGaussian2D(center, covariance);
    // Return the glossy motion vector.
    return prevX - X;
}
```

The usage of our stochastic glossy reflection motion vectors is similar to the shadow motion vectors. Also, the "natural hierarchy" still exists, i.e., when the roughness is high, the glossy reflection motion vectors will be more noisy, but the temporally filtered result will be then spatially cleaned up in a more aggressive manner.

We show the *sun temple* scene in Figure 25-4, which contains glossy reflections of various objects. We compare our method with four approaches: (1) using the traditional reflectors' motion vectors but without clamping of previous pixel values, (2) using traditional motion vectors with clamping, (3) using traditional motion vectors with the temporal component of A-SVGF (the temporal gradient method), and (4) using specular reflected rays' hit points' motion vectors, also with clamping. For all the comparisons, we use the spatial filter discussed in [6] as the spatial component of our denoising pipeline.

The comparison indicates that our glossy reflection motion vectors do not introduce ghosting artifacts. However, with traditional motion vectors, naive filtering produces significant ghosting. Clamping relieves the lagging but introduces severe discontinuous artifacts. With specular motion vectors, the results look plausible in most regions, but discontinuous artifacts can still be found around the edges of the reflected objects. The A-SVGF will always result in noisy results because the temporal gradient changes so fast that it

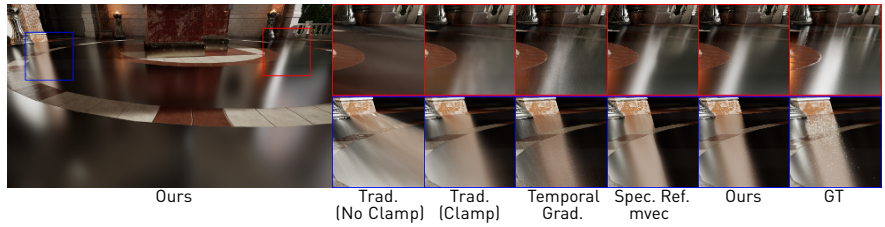


Figure 25-4. *The sun temple scene with a rapidly moving camera. Our stochastic glossy reflection motion vector is able to produce accurate reflections, whereas the traditional motion vectors result in significant ghosting artifacts.*

mostly uses only the noisy current frame. Our method produces the closest result to the ground truth (ray traced without firefly removal, thus always looks brighter).

25.3.3 OCCLUSIONS

Different from the previous cases on shadows and glossy reflection, when occlusion happens, in theory there are no temporal correspondences X^0_{i-1} of the pixels X_i appearing from the previously occluded regions. The back-projected motion vectors of these pixels will always land on the occluders, thus the previous pixel values cannot easily be used.

To alleviate this issue, we start from the clamping technique by Salvi [8] in TAA methods. Our insight is that if the color value at X^0_{i-1} is closer to the value at X_i, the issues produced by the clamping method can be better resolved. Also, close color values often appear on the same object. Inspired by Bowles et al. [1], we propose a new motion vector for the just-appeared region to find a similar correspondence in the previous frame. To do that, we refer to the relative motion.

As Figure 25-5a shows, the traditional motion vector gives the $X_i \rightarrow Y$ correspondence, but, unfortunately, cannot be easily used. Our method continues to track the movement of $Y \rightarrow Z$ from the previous frame to the current, using the motion of the occluder. Then, based on the relative positions of X_i and Z, we are able to find the location X^0_{i-1} in the previous

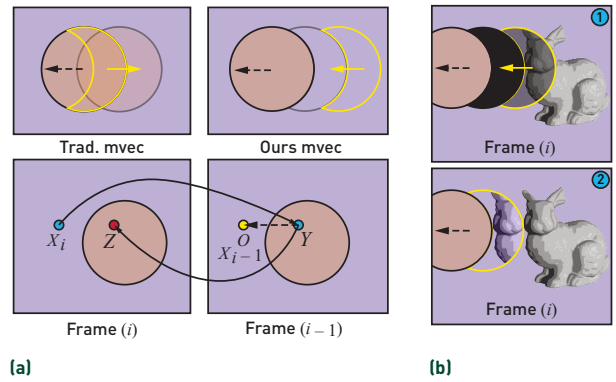


Figure 25-5. *(a) The computation of our dual motion vectors for occlusions, and comparison with traditional motion vectors. (b) How the repetitive pattern is produced when simply reusing colors with our dual motion vectors.*

frame. This process can be simply represented as

$$X^{O}_{i-1} = Y + (X_i - Z), \tag{25.6}$$

where $Y = \mathbf{P}_{i-1}\mathbf{T}^{-1}(X_i)\mathbf{P}_i^{-1}X_i$ and $Z = \mathbf{P}_i\mathbf{T}(Y)\mathbf{P}_{i-1}^{-1}Y$.

Equation 25.6 indicates that we have applied a back-projection followed by a forward-projection, essentially using two motion vectors. Thus, we name our approach *dual motion vectors* for occlusions. In this way, we are able to find a correspondence X^{O}_{i-1} with a much closer color value to X_i. Note that because we use $\mathbf{P}$ of two frames to track X_i, we are able to support the movement of the camera and objects simultaneously. The pseudocode for calculating our occlusion motion vector looks as follows:

```
calcOcclusionMotionVector(uint2 pixelIndex, float4 hitPos, uint hitMeshID,
     uint occluderMeshID){
    // For each pixel X_i that previously occluded ...
    float2 X = pixelIndex + float2(0.5f, 0.5f);
    // ... find Y on the occluder in the previous frame.
    float4 S = hitPos;
    // The custom function getT(...) returns the geometry transformation for
         a given mesh, and inverse(...) inverts a 4x4 matrix.
    float4 prevS = mul(S, inverse(getT(hitMeshID)));
    // The custom function toPrevScreen(...) projects a given world-space
        point to screen space to get the corresponding image-space pixel
        for the previous frame.
    float2 Y = toPrevScreen(prevS);
    // Then, find Z on occluder in the current frame.
    // The custom function getPrevHitPos(...) returns the world-space
        shading point for a given pixel in the previous frame.
    float4 SY = getPrevHitPos(Y);
    float4 curSY = mul(SY, getT(occluderMeshID));
    // The custom function toScreen(...) projects a given world-space point
        to screen space to get the corresponding image-space pixel.
    float2 Z = toScreen(curSY);
    // Return the occlusion motion vector.
    return Y + X - Z;
}
```

Note that because we deal with different effects separately for shadows and glossy reflection, and the shading part can already be reasonably approximated in a noise-free way (e.g., using the Linear Transformed Cosines (LTC) method [5]), we only have to apply our dual motion vectors to indirect illumination. Moreover, as glossy indirect illumination has been elegantly addressed using our glossy reflection motion vectors, we can now focus only on diffuse materials.

DISCUSSION: REUSING COLOR VERSUS INCIDENT RADIANCE As Figure 25-5b indicates, simply applying the color values as in Bowles et al. [1], using the dual motion vector will result in a clear repetitive pattern because it

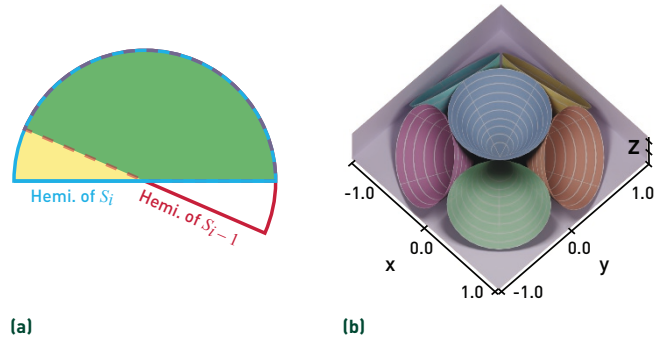


Figure 25-6. *(a) Our partial temporal reuse scheme. Only the same directions in the overlapped solid angle will share temporally coherent radiance. (b) We use six slightly overlapping cones on a hemisphere to store incident radiance approximately.*

is essentially copying and pasting image contents. This is especially problematic when the normals at S_i and S^0_{i-1} are different. Removing the textures from colors (i.e., *demodulation*) does not help because when the normals differ, the intensity of the shading result can differ significantly.

To address this issue, we propose to temporally reuse the incident radiance instead of the shading result. Specifically, for the application of diffuse indirect illumination, we record the 2D indirect incident radiance per pixel.

Figure 25-6a shows an example. Suppose that we have recorded the incident radiance of S_i and S^0_{i-1}, each on a hemisphere; then, we immediately know that all the directions in the overlapped regions of these two hemispheres (marked as green) could be temporally reused, while the non-overlapping part (marked as yellow) should remain using only the spatial content from frame i. Then, the radiance from both parts will be used to re-illuminate the shading point X_i, leading to an accurate temporally accumulated shading result.

For storing and blending the incident radiance, we refer to the representation in the voxel cone tracing approach [3]. We subdivide a hemisphere into six slightly overlapping cones with equal solid angles $\pi/3$ pointing in different directions, and we assume that the radiance remains constant within each cone. During spatial or temporal filtering, instead of averaging the shading result, we filter for each cone individually, using the overlapping solid angles between each pair of cones as the filtering weight. Figure 25-6b illustrates our idea. Finally, we are able to achieve a much cleaner result for diffuse indirect illumination.

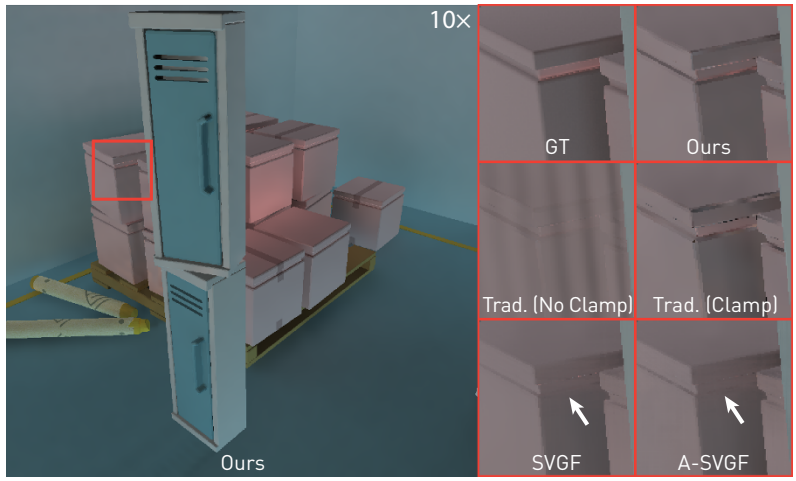


Figure 25-7. *A view of the PICA scene with objects moving from left to right.*

We demonstrate the effectiveness of our occlusion motion vectors in the *PICA* scene with moving objects in Figure 25-7. Only indirect illumination is shown, and its intensity is scaled ten times for better visibility. New unoccluded regions will appear around the boundaries of foreground objects. In these regions, the temporal information will simply be rejected with traditional motion vectors. Therefore, in this case the SVGF still results in a significant amount of overblur, whereas A-SVGF again appears to be smeared spatially and loses temporal stability. The clamping approach tries to use pixel values from the occluders; however, because the pixel values on the foreground and background usually differ drastically in the occlusion case, this will still introduce ghosting artifacts.

25.4 PERFORMANCE

Figure 25-8 shows the average computation cost of each step of denoising individual effects using our motion vectors. The average cost of an individual step was estimated from 500 frames rendered at 1920×1080 on an NVIDIA TITAN RTX. Compared with SVGF (traditional motion vectors), it can denoise different effects with a similar cost, around 3.11 ms per frame.

As one would expect from the simple computation in Section 25.3, in practice we observed only a negligible performance cost by replacing with our motion vectors, which is always less than 0.23 ms. Besides, we have also noticed that our implementation of denoising shadows and glossy reflections is already

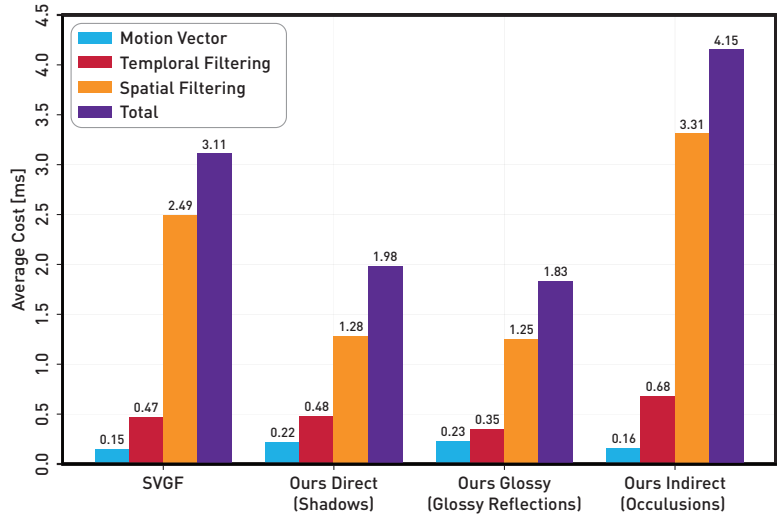


Figure 25-8. *Runtime breakdown of our methods.*

much faster than SVGF, as we use much simpler spatial filters and fewer levels or passes. Finally, when denoising indirect illumination, it is worth mentioning the additional cost when we introduce the cones for storing and filtering the incident radiance. For this, compared with reusing colors, we found that the typical cost of denoising each frame increased around 1.5 ms.

25.5 CONCLUSION

In this chapter, we have proposed multiple types of motion vectors for better utilization of temporal information in real-time ray tracing. With our motion vectors, we are able to track the movement of shadows and glossy reflections and to find similar regions to blend with previously occluded regions. We showed that our motion vectors are temporally more reliable than traditional motion vectors and presented cleaner results compared to the state-of-the-art methods with negligible performance overhead.

For more information about implementation details, comparisons, and limitations, we refer readers to our paper [11] and the accompanying video.

REFERENCES

[1] Bowles, H., Mitchell, K., Sumner, R. W., Moore, J., and Gross, M. Iterative image warping. *Computer Graphics Forum*, 31(2pt1):237–246, 2012. DOI: 10.1111/j.1467-8659.2012.03002.x.

[2] Chaitanya, C. R. A., Kaplanyan, A. S., Schied, C., Salvi, M., Lefohn, A., Nowrouzezahrai, D., and Aila, T. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics*, 36(4):98:1–98:12, 2017. DOI: 10.1145/3072959.3073601.

[3] Crassin, C., Neyret, F., Sainz, M., Green, S., and Eisemann, E. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum*, 30(7):1921–1930, 2011. DOI: 10.1111/j.1467-8659.2011.02063.x.

[4] Fernando, R. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, page 35, 2005. DOI: 10.1145/1187112.1187153.

[5] Heitz, E., Dupuy, J., Hill, S., and Neubelt, D. Real-time polygonal-light shading with linearly transformed cosines. *ACM Transactions on Graphics*, 35(4):411:–41:8, 2016. DOI: 10.1145/2897824.2925895.

[6] Liu, E., Llamas, I., Kelly, P., et al. Cinematic rendering in UE4 with real-time ray tracing and denoising. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 289–319. Apress, 2019.

[7] Mara, M., McGuire, M., Bitterli, B., and Jarosz, W. An efficient denoising algorithm for global illumination. In *High Performance Graphics*, pages 3–1, 2017.

[8] Salvi, M. Anti-aliasing: Are we there yet? Open Problems in Real-Time Rendering, SIGGRAPH Course, 2015.

[9] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, page 2, 2017.

[10] Schied, C., Peters, C., and Dachsbacher, C. Gradient estimation for real-time adaptive temporal filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(2):24:1–24:16, 2018. DOI: 10.1145/3233301.

[11] Zheng, Z., Shiqiu, L., Jinglei, Y., Lu, W., and Ling-Qi, Y. Temporally reliable motion vectors for real-time ray tracing. *Computer Graphics Forum (Proceedings of Eurographics 2021)*, 40(2):79–90, 2021. DOI: 10.1111/cgf.142616.

[12] Zimmer, H., Rousselle, F., Jakob, W., Wang, O., Adler, D., Jarosz, W., Sorkine-Hornung, O., and Sorkine-Hornung, A. Path-space motion estimation and decomposition for robust animation filtering. *Computer Graphics Forum*, 34(4):131–142, 2015. DOI: 10.1111/cgf.12685.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 26

RAY TRACED LEVEL OF DETAIL CROSS-FADES MADE EASY

Holger Gruen
Intel Corporation

ABSTRACT

Ray tracing techniques in today's game engines need to coexist with rasterization techniques in hybrid rendering pipelines. In the same throw, level of detail (LOD) techniques that are used to bring down the cost of rasterization need to be matched by ray traced effects to prevent rendering artifacts. This chapter revisits solutions to some problems that can arise from combining ray tracing and rasterization but also touches on methods for more generalized ray traced LOD transitions.

26.1 INTRODUCTION

The DirectX Raytracing (DXR) API [7] has incited a new wave of high-quality effects that replace rasterization-based effects. As of the writing of this text, fully ray traced AAA games are the exception. Even high-end GPUs struggle with ray tracing the massive amounts of dynamic geometry that some AAA games require. As a result, hybrid rendering pipelines that mix and match ray tracing and rasterization are commonplace.

Games engines employ a variety of techniques (see [6]) to reduce the geometric complexity of scene elements to bring down rendering cost. In order to avoid the complexity and limitations of continuous geometry decimation through vertex animation and the resulting edge collapse (also known as geomorphing, see [3]), many game engines instead cross-fade or cross-dither between two discrete geometry LODs (see, e.g., [2]), e.g., between LOD0 and LOD1. Here, LOD0 denotes a geometry that is comprised of fewer vertices and triangles than LOD1.

Typically, a transition factor f in the interval [0.0, 1.0] is used to control cross-faded LOD transitions (see, e.g., the ray traced transition example in Figure 26-1). During rasterized transitions, game engines need to draw both LOD0 and LOD1. When drawing LOD0, the pixel shader then uses a uniform

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_26

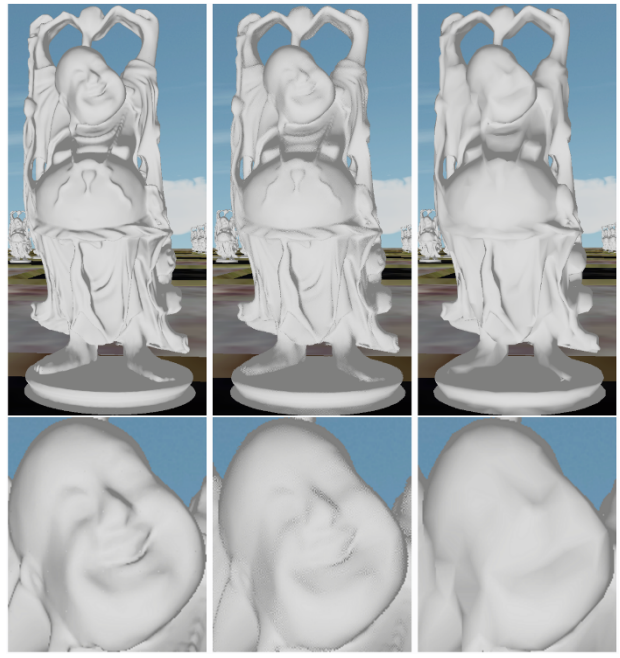


Figure 26-1. *Three stages of a ray traced LOD cross-fade. The highest LOD is shown on the left, a half-way transition is shown in the middle, and the lowest LOD is shown on the right.*

pseudo-random number r (from the interval [0.0, 1.0]) and discards the current pixel if $r < f$. The pixel shader that draws LOD1 uses the same pseudo-random number r and discards the current pixel if $r \geq f$.

This implies either that r is computed using the 2D position of a pixel as a seed or that a screen-space aligned texture containing random numbers is used. Low-discrepancy pseudo-random number sequences (see, e.g., [1]) or precomputed textures that store such sequences help to make the transition less noticeable.

Cross-faded LOD transitions are also a good choice for ray tracing applications as they prevent bounding volume hierarchy (BVH) refitting or rebuild operations for geomorphing geometries. Instead, bottom-level acceleration structures (BLASs) for discrete LODs can be instanced in the top-level acceleration (TLAS) structure as needed. So, similar to the fact that rasterization needs to render two LODs during transitions, it is necessary to put the geometry of two or more LODs in the BVH in order to enable ray traced transitions (see also [5]).

At this point in time, DXR doesn't provide obvious direct API support for letting potential traversal hardware handle high-quality LOD transitions, though.

The DirectX specification (see [7]) mentions *traversal* shaders (see [4]) as a potential future feature, but it is unclear if and when they will become a reality. In a hybrid ray traced technique, where the ray origin is usually derived from the world-space position of a given pixel, a traversal shader would allow LOD-based cross-fading using the same logic that the pixel shaders described previously use. Instead of discarding pixels, the ray would just be forwarded into the BLAS of the selected LOD according to the per pixel uniform random number r.

NVIDIA's developer blog (see [5]) describes a technique that uses the per-instance mask of a BLAS instance along with an appropriate ray mask to implement LOD cross-dithering that is accelerated by the traversal hardware. In this technique, the transition factor f is mapped to two different instance masks, e.g., *mask*0 for LOD0 and *mask*1 for LOD1. If, e.g., $f = 1.0$, all bits in the instance mask for LOD1 are set to 1 and no bits are set in the instance mask for LOD0. A transition factor of $f = 0.6$ means that *mask*1 has the first $uint(0.6 * 8)$ bits of the instance mask for LOD1 set to 1. The instance *mask*0 for LOD0 is then set to $\sim$ *mask*1&0*xFF*. This approach ensures that a per-pixel ray mask can be computed by randomly setting only one of its eight bits.

As Lloyd et al. [5] describe, their use of the 8-bit masks limits the number of levels for stochastic LOD transitions. Also, employing a transition technique that utilizes instance masks blocks these instance masks from being used for other purposes. If the limited number of LOD transitions turns out to be a quality issue for your application or if the mask bits are needed otherwise, it is possible to move the transition logic to the *any-hit* shader stage. The any-hit shader can evaluate stochastic transition without the limits described in [5]. It can store r in the ray payload. The any-hit shader can then ignore hits if $r < f$ and a triangle in the BLAS for LOD0 is hit. In the same manner, it can ignore hits for the BLAS for LOD1 if $r \geq f$.

However, using the any-hit shader stage for LOD cross-fades has a much higher performance impact than using the technique from [5] as it delays the decision to ignore a hit to a programmable shader stage that needs to run on a per-hit/triangle basis. Shader calls interrupt hardware traversal and can have a significant performance impact. Please note that a potential traversal shader implementation would also need to interrupt hardware traversal,

though not at the frequency of triangle hits but at the much lower frequency of hitting traversal bounding boxes.

In general, in situations where more than two LODs of an object are part of the BVH and the programmer wants to pick and chose between a number of LODs (e.g., purely based on the distance from the ray origin), any-hit or traversal shaders are the only tools of choice, as instance masks don't allow for this degree of flexibility.

26.2 PROBLEM STATEMENT

As outlined in the prevoius section and adding more details to the description in [5], in hybrid rendering pipelines ray traced techniques need to be able to manage the fact that rasterized G-buffers may represent different LODs of the same cross-faded object in neighboring pixels.

Adding, e.g., hard ray traced shadows to an engine that uses rasterization to lay down a G-buffer can be done like this:

1. Reconstruct the world-space position *WSPos* for the current pixel from the value in the depth buffer.
2. Compute the starting point of the shadow ray by offsetting *WSPos* along the unit-length per-pixel normal **N** that has been scaled by a factor *s*: $ray.Origin = WSPos + s * N$.
3. Trace the ray from the origin toward the light source.

The normal scaling factor *s* is chosen in a way that prevents self-shadowing (Lloyd et al. [5] call these self-shadowing artifacts *spurious shadows* if they result from mismatching LODs) but also prevents the localized loss of shadows from small local details. Assuming that the current G-buffer pixels contain some LOD cross-faded object, as depicted in Figure 26-2, then neighboring pixels may well belong to different LODs.

It is possible, depending on how the simplified LODs are built, that the geometry of LOD1 locally is farther out when compared to the geometry of LOD0 or vice versa, as shown in Figure 26-3.

As game engines currently place either the geometry of LOD0 or the geometry of LOD1 in the BVH, this can create problems. Assume that LOD1 is outside of the geometry of LOD0 and that only the geometry of LOD1 has been placed in the BVH. Under this scenario, a shadow ray that emanates from the

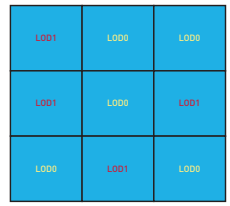


Figure 26-2. *A 3 × 3 portion of a G-buffer that contains pixels that have been rasterized from geometry of LOD0 and LOD1.*

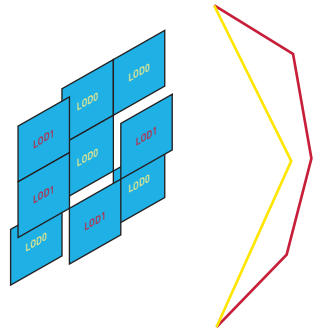


Figure 26-3. *A 3 × 3 portion of a G-buffer with pixels of LOD0 and LOD1 and a depiction of the underlying geometry of LOD0 in yellow and LOD1 in red.*

central G-buffer pixel (see Figure 26-2) and that belongs to LOD0 may well hit the geometry LOD1. The result is that this pixel is falsely assumed to be in shadow, as shown in Figure 26-4. Please note that which LOD is outside may well change across a model, so it isn't possible to pick the most suitable LOD from an engine point of view to prevent this.

It is, of course, possible to work around this by increasing the normal scaling factor *s* to a point that prevents this problem for all scene elements. But, as described previously, this may well lead to the loss of local details, in ambient occlusions or shadows. Also, increasing *s* raises the probability of moving the ray origin into some close-by geometry. Similar problems exist with almost all other ray traced techniques that need to work from a LOD-dithered G-buffer.

26.3 SOLUTION

The following steps help to work around the problems described above by making sure that rasterization and ray tracing use the same random number to implement LOD cross-fading based on an individual per-object transition factor *f*:

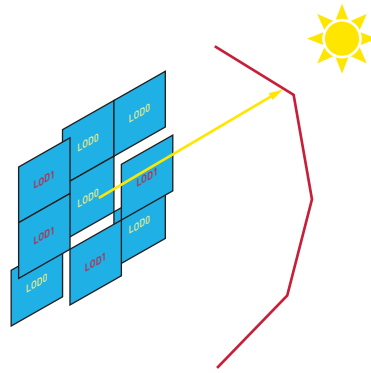


Figure 26-4. *Choosing the wrong LOD (e.g., LOD1) can lead to self-shadowing artifacts.*

1. During rasterization, pick a uniform pseudo-random number r from the interval [0.0, 1.0] that purely depends on the 2D position of the current pixel, or that depends on a combination of the 2D position and the instance ID or object ID of the current object.

   > If you intend to use any-hit or traversal shader–based ray traced transitions, then do the following:

   - When drawing LOD0, the pixel shader discards the current pixel if $r < f$.
   - The pixel shader that draws LOD1 discards the current pixel if $r \geq f$.

   > If you intend to use instance mask–based transitions (see [5]), then do the following:

   - Compute a binary $mask1$ that has the first $uint(f * 8)$ bits set to one to be used by the pixel shader for LOD1: $mask1 = (1 << uint((8 + 1) * f)) - 1$.
   - Compute $mask0$ to be used by the pixel shader for LOD0: $mask0 = (\sim mask1)\&0xFF$.
   - Multiply r by 7 to arrive at the bit index to compute $rmask$ to be used by the pixel shader for LOD1: $rmask = 1 << uint(r * 7)$.
   - When drawing LOD0, the pixel shader discards the current pixel if $(rmask\&mask1) == 0$.
   - The pixel shader that draws LOD1 discards the current pixel if $(rmask\&mask0) == 0$.

   Please note that an instance mask–based transition may limit the quality of a rasterized cross-fade in the same way as it may limit ray

traced LOD transitions. If quality is a concern for you, you may need to resort to any-hit shader–based transitions as outlined in Section 26.1.

2. Put BLAS instances for LOD0 and LOD1 of all objects into the TLAS that are currently undergoing a LOD transition.

3. Use the same uniform random number r that was used during rasterized cross-fading in your ray traced cross-fading setup for the current pixel.

   - For an any-hit (or a future traversal) shader–based transition, put r into the ray payload and ignore hits if $r < f$ and a triangle in the BLAS for LOD0 is hit. In the same vein, ignore hits for the BLAS for LOD1 if $r \geq f$. Local root signature constants in the shader binding table for the hit entries for LOD0 and LOD1 can be used to detect if a triangle from LOD0 or LOD1 has been hit.
   - For the case of instance mask–based transitions, as described in [5], do the following:
     - Compute a binary $mask1$ that has the first $uint(f * 8)$ bits set to one to be used by the pixel shader for LOD1: $mask1 = (1 << uint((8 + 1) * f)) - 1$.
     - Compute $mask0$ to be used as the instance for LOD0: $mask0 = (\sim mask1)\&0xFF$.
     - When tracing a ray, use $uint(f * 7)$ to set one random bit in the ray mask $rmask$: $rmask = 1 << uint(r * 7)$.

   Using the approach outlined here, the rays cast from the reconstructed world-space positions of a pixel always only return hits with the same LOD that was used when rendering the pixel.

26.4 FUTURE WORK

Animating a LOD cross-fade shows that there is a clear difference in the fluidity of the transition between a mask–based fade and a comparison value–based fade using an any-hit shader. It would be beneficial to extend future ray traversal hardware to include a per-BLAS instance comparison value and a flag indicating either a less than or a greater than comparison value, which then gets stored in the BVH. The `TraceRay` intrinsic could be extended to add an additional 8-bit comparison value parameter as well.

The traversal hardware could then, on top of the binary `AND` operation on the mask, perform the comparison operator on the values of the instance and the

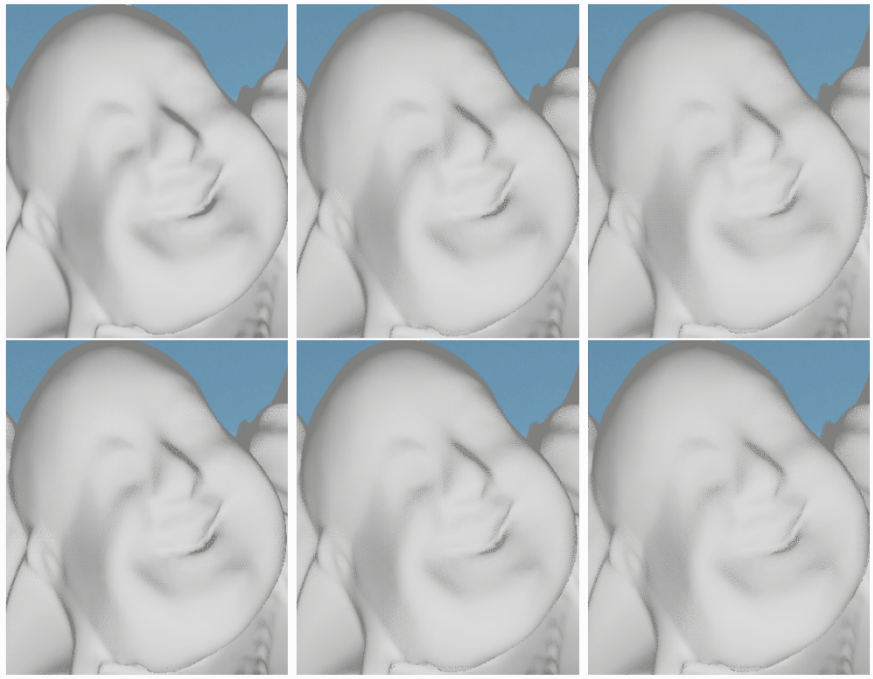


Figure 26-5. *Three stages of a ray traced LOD cross-fade at f = 0.904 (left), f = 075 (middle), and f = 0.647 (right). Top: images produced using a mask-based transition. Bottom: Using an emulated comparison value–based transition.*

value of the ray. Ray traversal into an instance could only continue if the comparison operations returns `true`.

This new functionality would enable hardware-assisted LOD cross-fades that are comparable in quality to what any-hit shader–based cross-fades produce today.

Figure 26-5 shows three images from a LOD transition using a mask-based and an emulation of a comparison value–based transition. While the full difference in quality and fluidity can only be shown in a video, please note that the comparison-based method produces transitions with many more intermediate steps than the mask-based transition.

26.5 CONCLUSION

This chapter expands on existing solutions that make sure that there are no mismatches between rasterized LOD and ray traced LOD. The method

described can make use of fully accelerated hardware traversal, if the quality of instance mask–based transitions is good enough to meet your quality requirements. If this is not the case, you will need to pick the slower path that involves calls to any-hit shaders for objects that are currently undergoing a LOD transition. Future traversal hardware that features comparison value–based conditional traversal can deliver high-quality and fluid LOD transitions at full speed without the use of any-hit shaders.

REFERENCES

[1] Ahmed, A. G. M., Perrier, H., Coeurjolly, D., Ostromoukhov, V., Guo, J., Yan, D.-M., Huang, H., and Deussen, O. Low-discrepancy blue noise sampling. *ACM Trans. Graph.*, 35(6), Nov. 2016. ISSN: 0730-0301. DOI: 10.1145/2980179.2980218. https://doi.org/10.1145/2980179.2980218.

[2] Arlebrink, L. and Linde, F. A study on discrete level of detail in the Unity game engine. https://draketuroth.files.wordpress.com/2018/06/a-study-on-discrete-level-of-detail-in-the-unity-game-engine.pdf, 2018.

[3] Hoppe, H. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 99–108, 1996. DOI: 10.1145/237170.237216.

[4] Lee, W.-J., Liktor, G., and Vaidyanathan, K. Flexible ray traversal with an extended programming model. In *SIGGRAPH Asia 2019 Technical Briefs*, pages 17–20, 2019. DOI: 10.1145/3355088.3365149.

[5] Lloyd, B., Klehm, O., and Stich, M. Implementing stochastic levels of detail with Microsoft DirectX Raytracing. https://developer.nvidia.com/blog/implementing-stochastic-lod-with-microsoft-dxr, June 15, 2020.

[6] Luebke, D., Reddy, M., Cohen, J., Varshney, A., Watson, B., and Huebner, R. *Level of Detail for 3D Graphics*. Morgan Kaufmann, 1st edition, 2012.

[7] Microsoft. DirectX Raytracing (DXR) functional spec. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html, 2021.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 27

RAY TRACING DECALS

Wessam Bahnassi
NVIDIA

ABSTRACT

This chapter discusses several different approaches to implementing decals in ray tracing. These approaches vary between the use of triangle meshes versus procedural geometry and support for single versus multiple decals per surface point. Additional features are also discussed for using decals in production, as well as a number of possible optimizations. The discussion is supported by performance measurements to give the reader an idea of the cost of the different ray tracing techniques, in hope that this inspires selection of the best technique whether for decals or other applicable ray tracing effects.

27.1 INTRODUCTION

Decals are a common element in 3D game scenes. They are mainly used to add detail over an existing scene in a similar effect to real-world decals: an image imprinted on top of an existing surface (see Figure 27-1). In a game, this can be detail over a rather repetitive wall pattern or holes resulting from bullets hitting an object. The texture from the decal replaces the underlying surface's texture prior to lighting. This effect has been employed for a long time and was achieved via different approaches for both forward (e.g., [1]) and deferred shading (e.g., [2]) renderers. In this chapter, the application of decals

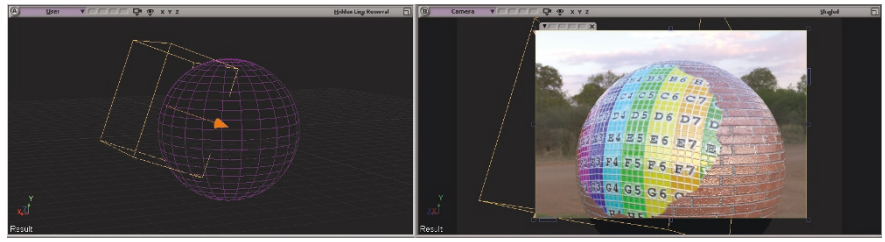


Figure 27-1. *A decal applied on a sphere in a ray traced scene, replacing both albedo and normals of the underlying surface.*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_27

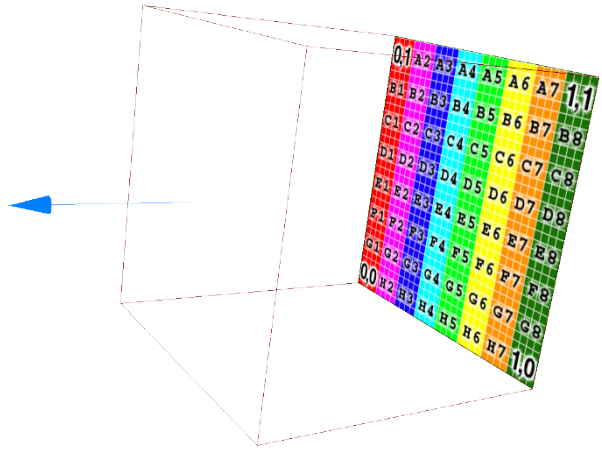


Figure 27-2. *A decal can be described by a box and a direction of projection. The box determines the coverage and size of the projected texture in the scene.*

in a DirectX Raytracing (DXR) rendering engine is discussed, with topics covering formulation, scene setup, shaders, and optimizations.

27.2 DECAL FORMULATION

Decals have been most commonly described by a texture and a directed box that specifies the location and orientation of the decal's projection. Any surfaces intersecting that box are expected to receive that decal. (See Figure 27-2.) Depending on the decal's purpose and properties, formulations other than the one described above might be preferable (as shown in Section 27.5.3). The goal is to transfer the decal's texture onto the target surface at the right time in the rendering pipeline. One possible way to achieve this is to render an overlay of the affected surface's triangles but using the decal texture instead of the base surface texture. This requires computing the intersection between the decal's box and the target geometry to generate a set of clipped triangles with texture coordinates that map the decal texture correctly. However, this is an old approach that suffers from performance and precision problems.

An alternative way that appeared with deferred shading renderers is to rasterize decal boxes after the G-buffer pass but prior to lighting. Pixels covered by the rasterized decal boxes execute a pixel shader that checks if the

world position actually falls within the decal box. If yes, then the shader computes the decal texture coordinates and samples the decal. The sampled values overwrite the G-buffer values, and the lighting pass is then carried out as usual. With this approach, it is easy to have decals modify G-buffer channels other than just albedo. For example, it is possible to have the decal overwrite normals and specular, thus allowing for more interesting effects than just color replacement, like material aging and dents.

This approach is valid also for hybrid deferred ray tracing renderers because in such renderers the G-buffer pass is still maintained from standard deferred shading, and the G-buffer is used to avoid tracing primary rays. However, ray traced reflections and surfaces need to use a different approach to apply decals to the ray tracing results. This is the focus of the rest of this chapter, now that the historic information is covered.

27.3 RAY TRACING DECALS

DXR divides the scene into primitives or meshes described by bottom-level acceleration structures (BLASs), and those primitives are instanced at one or more locations in the scene. The top-level acceleration structure (TLAS) stores the information for those instances. In addition to triangle meshes, the BLAS is also able to describe procedural primitives as simple axis-aligned bounding boxes (AABBs) that encompass procedurally evaluated shapes using an intersection shader during ray tracing.

The DXR representation can also carry a few flags that specify additional properties of the object. Of particular interest is the `OPAQUE` flag, which can be specified on both DXR scene levels (BLAS and TLAS).

Under DXR 1.0 (i.e., `DispatchRays()`), when a primitive is hit by a ray, it is possible to use local root signatures to drive DXR to bind primitive-specific resources automatically during the execution of the closest-hit or any-hit shaders. However, such facility is not available with DXR 1.1's `TraceRayInline()` style of ray tracing, so resource access is usually done manually via global resources and resource arrays.

In this chapter, we chose to store the information about each decal in a structure, and all decals are made available to our shaders by grouping them in a `StructuredBuffer`. Decals can be identified by their index in this `StructuredBuffer`, and their properties can be fetched when needed.

The following code shows an example of a decal structure that supports replacing albedo and normal textures of a surface:

```
struct Decal
{
    // World space to "unit box" space
    float4x3 world2DecalUnit;

    // Index in array of textures (-1 means unused)
    int albedo, normal;
};
StructuredBuffer<Decal> scene_decals : register(t0);
```

Following the decal formulation described earlier, an affine transformation matrix can be built such that it transforms a unit box (–0.5, –0.5, –0.5) – (0.5, 0.5, 0.5) to a world-space box of certain dimensions, orientation, and placement. This world-space box represents the decal's projection box in the scene. The inverse of the aforementioned matrix is what is stored in the `world2DecalUnit` member in the code listing. To sample a decal, the receiving surface point (in world space) must be transformed using the `world2DecalUnit` matrix. If the point was indeed inside the decal box, then its transformed XYZ coordinates must fall within the range [–0.5, 0.5].

To generate [0, 1] UV coordinates for sampling the decal texture at the surface point, we add 0.5 to the transformed `x` and `y` coordinates and pass the values as UV coordinates for texture sampling. This additional step can be merged into the `world2DecalUnit` matrix too, but the point-in-decal check must be adjusted accordingly as well because the valid range for `x` and `y` coordinates becomes [0, 1].

The next section focuses on finding which decals project onto which points in the scene.

27.3.1 TRACING ONE DECAL

Given a point on a surface in the scene, the question is to know whether there is a decal box covering that point. This is in fact a point-in-volume question, which neither the DXR 1.0 nor DXR 1.1 API natively support. The following sections discuss approaches to answer the point-in-volume question.

POINT-IN-VOLUME BY INTERSECTING AGAINST DECAL BOX FACES

One way is to cast a ray from the surface point and check if it hits the inner face of any decal box. To achieve this, the scene must be set up as follows.

A decal mesh is built as an axis-oriented unit box (edge length is 1) made of 12 triangles. The triangles winding order must be consistent such that it is possible to identify the inner side versus the outer side of the box. For example, let us use clockwise winding order and make the triangle faces point outside the box. Assume the Z+ axis of this unit box to be the direction of projection of the decal.

A BLAS is built for this mesh. To instantiate decals in the scene, the same decal BLAS is always referenced for each added TLAS instance entry. Each decal instance uses a transformation matrix that scales, rotates, and positions the decal unit box mesh to match the required placement.

The decal instances must carry a user-defined mask value to allow tracing against them without hitting other entities in the DXR scene (the `InstanceMask` member of `D3D12_RAYTRACING_INSTANCE_DESC`). The `InstanceID` field should be mappable to an index in the decal `StructuredBuffer` such that decal information can be accessed once it is returned by ray tracing.

To answer the point-in-decal question, a DXR ray query is defined to originate from the surface point in question and go in some direction, say (0, 0, 1). The ray length can be set to the diagonal of the largest decal in the scene. The query ignores frontfaces and uses an instance mask to only trace against the decals in the scene instead of tracing all scene entities. The closest-hit shader will be invoked for the decal encompassing the point (if any), where the decal index can be reported back through the ray payload.

Unfortunately, relying directly on the result returned from this setup can lead to false positives, as the ray query can hit the inside of a decal box even if that box does not encompass the point in question (see Figure 27-3 for such a case). Thus, the returned payload result must be validated by checking if the surface point falls within the reported decal box. If not, the ray must be continued until its length is exhausted. It is possible to rely on an any-hit shader to do the same validation, which means that the closest-hit shader's result will always be valid. The any-hit shader will be called for each inner decal box face the ray hits, and the any-hit shader will only report a hit if the decal box is validated to contain the surface point. It is important to note that the decals must avoid using the `D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQUE` and `D3D12_RAYTRACING_INSTANCE_FLAG_FORCE_OPAQUE` flags in the BLAS and the TLAS, otherwise the any-hit shader will not be invoked.

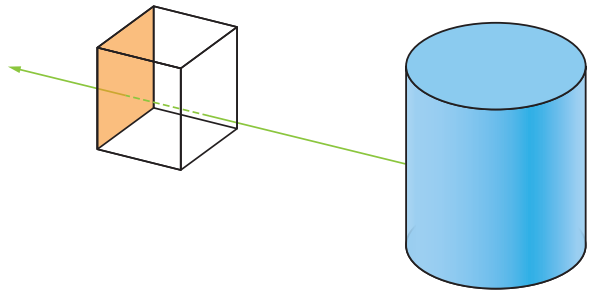


Figure 27-3. *A decal wrongly identified as encompassing a surface because the ray from the surface point hit the inside face of the decal box. This test alone is not enough to give the correct containment answer.*

POINT-IN-VOLUME BY INTERSECTION SHADERS

A different way to answer the point-in-volume query is to use a DXR intersection shader. As described earlier, intersection shaders are needed for procedural (AABB) DXR primitives. The key is that DXR will have to execute the intersection shader for any AABB that intersects *or contains* the ray in the query.

For this to work, two key changes must be made to the previous setup.

First, the scene TLAS must switch to use a procedural BLAS instead of the triangle mesh BLAS described earlier. The procedural BLAS AABB is simply set to have the extents of [–0.5, –0.5, –0.5] – [0.5, 0.5, 0.5]. This time, the procedural BLAS can use `D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQUE`. The `InstanceMask` and `InstanceID` in the TLAS are filled the same way as in the previous section.

The second key change is the hit group given to the decal instance entries in the TLAS. The hit group must reference a `ProceduralPrimitiveHitGroup` that uses both a closest-hit shader and an intersection shader.

The intersection shader does not actually have to do any calculations, as our decal box matches exactly the procedural BLAS AABB. Thus, if the intersection shader was invoked, then the decal box must have been accurately hit by the ray indeed. Thus, all what the intersection code will do is simply `ReportHit(RayTCurrent(), 0, attr);`.

The closest-hit shader will receive the reported hit and directly fill the payload with the hit decal ID (using the `InstanceID()` intrinsic).

Because the intersection shader will be called even if the ray is fully included inside the procedural primitive's AABB, the ray query can now be set to use a very short ray, originating from the surface point. For example, the `RayDesc`'s `Direction` member can be set to (0, 0, 1), `TMin` set to 0, and `TMax` set to a tiny value (e.g., 0.00001). It is also possible to use the `RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH` flag as we are looking for one decal at most.

This approach is more elegant and performs much faster than the previous one because the answer (either negative or positive) will be definitive from one ray query without the need for shader-side loops. Thus, this approach is recommended if the rendering engine is fine with supporting a single decal on a surface at any time. If multiple decals are expected to overlap, not only will this approach fail to properly compose the correct results, but also the results could be unpredictable, leading to flickering and noise as the ray query returns any of the decals surrounding the same point.

To support multiple decals, more work is needed, and this is what the next section will explain.

27.3.2 TRACING AND BLENDING MULTIPLE DECALS

When a single decal on a surface point is not enough for artistic reasons, it is possible to extend support for blending multiple decals over the same point. This added complexity will come with a cost, obviously, which is why support for a single decal was described in detail previously. The added cost might not be worth it, and a single decal would have to make due.

A big challenge in supporting multiple decals is the order of processing the decals. The standard "replace" blending formula is not commutative, thus decals must be blended with the underlying surface color in the correct order.

Next, two methods are discussed to support multiple decals with correct blending order.

LIMITED SORTED DECALS

It is possible to extend the procedural AABBs technique described in Section 27.3.1 to support more than a single decal while hopefully benefiting

from the technique's performance advantage over other methods. The only catch is that a maximum number of decals per point must be established, and the reason for this limitation will become apparent later in the discussion.

The first change is on the geometry flags:

- Avoid using the `D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQUE` and `D3D12_RAYTRACING_INSTANCE_FLAG_FORCE_OPAQUE` flags in the BLAS and the TLAS.

- Specify `D3D12_RAYTRACING_GEOMETRY_FLAG_NO_DUPLICATE_ANYHIT_INVOCATION` on the procedural BLAS.

The second change is in the hit group setup. The procedural hit group must now use an any-hit shader instead of a closest-hit shader, and the ray query can use the `RAY_FLAG_SKIP_CLOSEST_HIT_SHADER` because it is not needed. Otherwise, the ray query remains the same tiny ray setup as before. Interestingly, even the `RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH` flag can still be used.

The any-hit shader will now be invoked for every decal box that contains the ray, where it has to properly collect the results and report them back to the ray query. To allow more invocations of the any-hit shader to occur, the shader must keep calling `IgnoreHit()` until the maximum allowed number of decals have been exhausted, then it should call `AcceptHitAndEndSearch()`.

If invocations to the any-hit shader were guaranteed to follow a particular order, then the any-hit shader can immediately sample the decal during its invocation and blend it with the payload. However, the calling order of the any-hit shaders is unpredictable under DXR, which forces us to have to collect the hits and sort them in able to respect the proper blending order.

If we limit the number of decals supported to a small maximum n (e.g., 2 or 3), then it is possible to fit an array of n decal IDs in the payload, which the any-hit shader can fill one by one in every invocation.

The benefits of keeping the maximum number small is that it also allows the returned results to be sorted in a fast manner even with simple $O(n^2)$ sorting algorithms.

Upon return of the ray query, the payload should now be filled with a number of decal IDs that are sorted, sampled, and blended in the correct order.

UNLIMITED SORTED DECALS

Another way to support multiple decals without maintaining sorted lists is to find a mechanism in DXR that already supports the ordering we need. It is possible to capitalize on closest-hit ray queries to guarantee ordered results.

We revert to the triangle mesh decal BLAS setup from Section 27.3.1, but some further changes are needed.

First, the decal mesh BLAS should specify the `D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQUE` flag.

Second, the ray query is modified to cull backfaces of the mesh (use the `RAY_FLAG_CULL_BACK_FACING_TRIANGLES` flag), and the ray's origin and endpoints are swapped. The ray now points *toward* the surface point rather than away from it and originates from the farthest location away from the surface point.

To blend the decals, the ray query is executed in a loop that exhausts the ray's length. For each closest hit returned, the decal is sampled and blended with the surface. Then, the ray is continued from the last hit. Once the ray length is exhausted or no further hits are found, all applicable decals will have been accumulated and blended in the correct order.

Even though this method is generic and can handle an unlimited number of overlapping decals correctly, it is much slower compared to the limited number of decals method due to the high overhead of casting rays repeatedly in a loop.

27.4 DECAL SAMPLING

Depending on the artistic requirements, a decal system may settle with simple texture blending (e.g., using the decal texture's alpha channel) or offer a more sophisticated material graph evaluation option. For the latter case, DXR supports callable shaders. Each decal can be associated with a callable shader that is called to evaluate the decal's channels (e.g., albedo and normal) and their blending values at the specified UV address computed from the projection of the decal over the surface point.

27.5 OPTIMIZATIONS

In this section, a few notes and ideas are provided to control performance for the different approaches mentioned in this chapter.

27.5.1 RAY LENGTH

For methods relying on a triangle mesh BLAS, it was suggested to use a ray length equal to the longest diagonal of any decal box in the scene. This value takes into account the worst case of a decal box's corner slightly touching the surface. The downside of this strategy is twofold: First, the CPU must keep track of decal box diagonal lengths and update the maximal length accordingly across the entire scene. Second, this maximal value might be driven by an outlier decal that is uniquely large across the rest of the decals in the scene, thus causing an increased ray length for all decal tracing even though most decals are way smaller than the outlier value. If this becomes a problem, the engine must find a better way to drive the decal ray lengths. For example, it might track decals that intersect each mesh and store the maximal decal diagonal length of those decals in the mesh's information. This way, the shader can use this value when casting decal rays from that mesh's surface, and meshes that are detected to have no decals on them can skip decal ray casts entirely.

27.5.2 SEPARATING THE TLAS

The decal queries are all supposed to ignore other entity types in the DXR scene. So far, the suggested way was to use instance masks to limit the ray query to decals only. However, another way is to place all decals in a separate DXR scene than the main scene. This is possible by building a TLAS that includes the decals only and binding it to a `RayTracingAccelerationStructure` variable in HLSL separate from the main scene's `RayTracingAccelerationStructure` variable. The decal ray queries use the decal TLAS instead of the main scene TLAS. The benefits of this separation are twofold: First, TLAS updates can be done separately from the main scene, allowing for parallelization on both CPU-side population and GPU-side building. Second, the rays iterating the decal TLAS do not have to go through a hierarchy imposed by unrelated scene elements (i.e., world meshes). This gives more decal ray casts a better chance to exit early, potentially even without touching any bounding volume hierarchy nodes.

The idea of moving decals to a separate TLAS also opens the door for more advanced use cases mentioned later.

However, there could be situations where this technique might actually hurt performance instead of improving it, and this is mainly due to cache thrashing caused by switching between two different TLAS data sets during ray tracing.

Thus, it is advisable to first experiment with this optimization with representative scenes and measure the gains before settling on it.

27.5.3 TIGHTER BLAS

Another potential optimization that affects all triangle mesh BLAS methods is to use tighter-fitting geometry for the decal contents. For example, if the decal texture's effective texels are mostly bound by a round shape (e.g., a round scorch mark), then a cylinder mesh BLAS is a better fit than a box mesh BLAS. Tighter shapes reduce processing of unnecessary texels and might outweigh the cost of a more complex BLAS for large area decals.

27.5.4 EVALUATION ORDER

Lastly, the way decals are described as being applied on top of surfaces might imply that they should be evaluated after evaluating the base surface's material. However, the evaluation order could be reversed. Decals can be evaluated before the underlying surface's material evaluation. This way, if the decal completely overrides a particular channel (e.g., albedo), then there is no use evaluating that channel for the underlying surface point's material. This optimization can hopefully balance the costs of decal evaluation in the scene. The only expensive parts of the decal left are those that partially blend with the surface, as both the decal and the surface material channels must be evaluated and blended.

27.6 ADVANCED FEATURES

During production, decal placement can sometimes become difficult due to scene density. The designer tries to adjust the decal box to only affect a certain mesh, but it ends up also touching other meshes in the same area. Or, consider a situation where a decal box placed in a certain location to decorate the scene is crossed by a moving object in the game. The moving object thus gets covered by the decal as well. Those cases often lead to an important feature in production decal systems, which is called *inclusion lists*. This is a list specified for each mesh in the scene, and it contains references to the only decals that are allowed to project on the mesh. Other decals must be ignored if they are not part of the mesh's decal inclusion list.

To support this feature in ray traced decals, it is possible to rely on the instance mask of the decal's ray query. However, the number of bits in this mask is quite limited and cannot scale to any scene made of even just a hundred objects. We mentioned before that it is possible to put decals in a

TLAS separate from the main scene TLAS. Now we take this idea and push it even further. The scalable solution for inclusion lists is to use a separate TLAS for each mesh's inclusion list. The decal TLASs for all meshes in the scene can be bound to the shader as an array of `RayTracingAccelerationStructures`.

For example, the first TLAS entry in the array contains all decals in the world, and subsequent TLASs in the array are specific to meshes with inclusion lists. Thus, it is enough for the mesh to carry its corresponding decal TLAS index in the shader and use that index for decal ray casting in that specific TLAS. The following code listing shows pseudocode illustrating this idea:

```
struct Mesh
{
    float4x3 object2World;
    ...
    // 0 if this mesh doesn't use an inclusion list
    uint decalSceneIndex;
};
RayTracingAccelerationStructure decalScenes[] : register(t10);

// ... During shading code of the mesh, trace decals that
// are applicable to this mesh only.
TraceRay(decalScenes[mesh.decalSceneIndex], ...);
```

27.7 ADDITIONAL NOTES

This chapter might have assumed DXR 1.0 shader table–style ray tracing for implementing decals. However, DXR 1.1 inline ray tracing can achieve the same functionality with some adjustments to the way rays are cast and to the processing of their results. DXR 1.1 inline ray tracing of decals is left as an exercise for the reader, but it is important to mention it now because it opens the door for the next note.

Deferred decals have been one of the major benefits of deferred shading renderers. With ray traced decals, forward shading renderers can also be extended to support decals conveniently and with the same scalability as deferred decals without relying on clustered approaches. The renderer simply casts a decal ray during the shading code of forward rasterization (using inline ray tracing), and that decal's evaluation overrides the input channels to the forward shader's lighting/material formula. Thus, it is possible to draw a mesh with decals and fully light it in one draw call.

27.8 PERFORMANCE

The different methods outlined in this chapter have been tested in a DXR scene to show the performance characteristics of each method. The scene is

	Combined TLAS	Dedicated TLAS
No decals	0.71 ms	0.71 ms
Single/Mesh/CHS	12.40 ms	12.73 ms
Single/Mesh/Any-hit	1.31 ms	1.28 ms
Single/AABB	0.81 ms	0.77 ms
Multi/Mesh/CHS	12.08 ms	11.05 ms
Multi (max 3)/AABB	0.84 ms	0.80 ms

Table 27-1. *Performance results collected on an NVIDIA RTX 3080 graphics card.*

the famous Sponza scene, which is made of triangle meshes comprising a total of 261,978 triangles. The test application spawns an excessive number of 36,582 decals and randomly scatters them over different surfaces of Sponza meshes. The intention behind this large number of decals is to highlight the cost of decals such that comparisons between the different decal tracing methods are easier to compare.

The sample application is available on the book's source code GitHub repository for experimentation.

Table 27-1 shows the performance numbers collected from stable runs on an NVIDIA RTX 3080 graphics card. The "No decals" entry represents the base scene without ray tracing decals, and the scene TLAS is only populated with Sponza meshes. The "Combined TLAS" column refers to timings collected using a single TLAS that contains instances for all meshes and decals together, where the instance mask is used to specify whether to trace meshes or decals in the scene. The "Dedicated TLAS" column represents timings collected by ray-tracing the scene using two different TLASs: one dedicated for scene meshes and another containing only decal instances as described in Section 27.5.2.

The row titles indicate which ray tracing method was used for decals. "Single" versus "Multi" refers to a how many decals per surface point are supported. "Mesh" versus "AABB" refers to the use of a triangle mesh BLAS for decal boxes versus a procedural box described by an AABB. "CHS" refers to methods using a loop around `TraceRay` where the result of a closest-hit shader is used. "Any-hit" indicates the method where the loop was replaced by the use of the any-hit shader.

The table shows a huge overhead associated with the two methods where the shader manually loops over `TraceRay` to collect decals. Those methods should be simply avoided. The AABB methods provide the best performance

and only add a small overhead on top of the base ray tracing scene cost. In less-exaggerated decal scenes, decal ray tracing performance should be barely noticeable.

Finally, the table shows interesting benefits when using a dedicated TLAS for decals. It is highly recommended to try this technique or extended versions of it as described in Section 27.6.

27.9 CONCLUSION

In this chapter, we have described decals and their formulation, and we have discussed several methods for supporting decals in ray traced scenes with different approaches like tracing box meshes and procedural primitives using intersection shaders. Depending on the rendering engine's needs, support for tracing and blending multiple decals was also presented. One of the common production features with decals is inclusion lists. This chapter discussed support for this feature while keeping efficiency and scalability as a target. Finally, the performance of the outlined techniques was measured in a sample application, and the results were shown for comparison, demonstrating the performance advantages of certain methods over others.

Hopefully, this chapter has provided enough information to guide you when adding decal support to your ray tracing rendering engine, as well as a few cool ideas that can be applied even outside the use of decals, but for real-time ray tracing in general.

REFERENCES

[1] Lengyel, E. Applying decals to arbitrary surfaces. In M. A. DeLoura, editor, *Game Programming Gems 2*. Volume 2, pages 411–415. Charles River Media, 2001.

[2] Persson, E. Volume decals. https://www.humus.name/index.php?page=3D&ID=83, October 4, 2009. Accessed January 19, 2021.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 28

BILLBOARD RAY TRACING FOR IMPOSTORS AND VOLUMETRIC EFFECTS

Felix Brüll, Robin Fynn Diedrichs, and Thorsten Grosch
Clausthal University of Technology

ABSTRACT

In rasterization, impostors and volumetric effects are often represented by billboards because they can be drawn faster than their 3D counterparts. This chapter explains how to render plausible reflections and refractions of billboards and presents two methods for rendering volumetric effects with DirectX Raytracing hardware.

28.1 INTRODUCTION

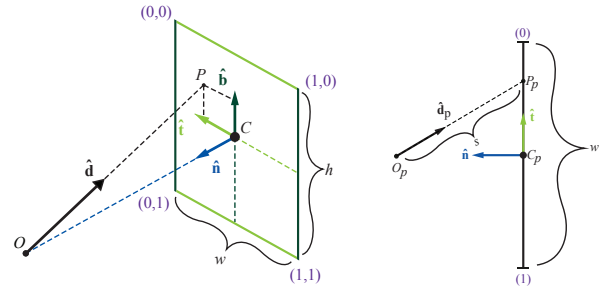
Traditional billboards are camera-oriented quadrilaterals defined by a position, width, height, and texture. Due to their simple shape, they can be drawn very fast and are easy to manage for animations. In rasterization, billboards always face the camera. However, this behavior is no longer correct after reflections and refractions.

We therefore describe how billboards can be used correctly in the context of ray tracing. In Section 28.2 we first explain how to implement impostor billboards for ray tracing. We also discuss how to determine the orientation of billboards dynamically after reflections and refractions. In Section 28.3 we then modify the billboards to act as smooth volumetric effects. Additionally, we present an alternative interpretation for volumetric billboards. Finally, we compare the performance of the presented techniques in Section 28.4.

28.2 IMPOSTORS

Impostors imitate important visual characteristics of other objects, but are faster to draw than their original [7]. Tree impostors are often represented by a single textured billboard. Even though more sophisticated impostor strategies such as billboard clouds [3] and octahedral impostors [13] exist,

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_28



(a) *World space* **(b)** *Local space* (C_p = **0**)

Figure 28-1. *Important properties and relations of the billboard, where O is the ray origin,* $\widehat{\mathbf{d}}$ *its direction, C is the center of the billboard with width w and height h, and P is the ray/billboard intersection point. The purple numbers represent the texture coordinates.*

this representation is often preferred if possible due to the very low rendering cost.

28.2.1 IMPLEMENTATION

We define an impostor billboard with its center C, its width w, its height h, and an optional axis alignment. For a ray with origin O and direction $\widehat{\mathbf{d}}$,[1] the billboard tangent space (see Figure 28-1a) can be computed with

$$\mathbf{n} = O - C, \quad \mathbf{t} = \widehat{\mathbf{u}} \times \mathbf{n}, \quad \mathbf{b} = \mathbf{n} \times \mathbf{t}, \tag{28.1}$$

with up-vector $\widehat{\mathbf{u}}$. If the billboard should be aligned to a specific axis, for example the up-axis $\widehat{\mathbf{u}} = (0, 1, 0)^\mathsf{T}$, the normal calculation needs to be adjusted:

$$\mathbf{n} = \left(O_x - C_x, 0, O_z - C_z\right).^\mathsf{T} \tag{28.2}$$

After obtaining the tangent space, the ray/plane intersection point P_p can be calculated after transforming the ray origin and direction into local billboard coordinates (see Figure 28-1b):

$$O_p = \left[\widehat{\mathbf{t}}\,\widehat{\mathbf{b}}\,\widehat{\mathbf{n}}\right]^\mathsf{T} \cdot (O - C), \quad \widehat{\mathbf{d}}_p = \left[\widehat{\mathbf{t}}\,\widehat{\mathbf{b}}\,\widehat{\mathbf{n}}\right]^\mathsf{T} \cdot \widehat{\mathbf{d}}, \tag{28.3}$$

$$P_p = O_p + s\widehat{\mathbf{d}}_p, \quad s \in \mathbb{R}. \tag{28.4}$$

As P_p is a point on the plane in plane coordinates, we can calculate s with

$$P_{pz} = 0 = O_{pz} + sd_{pz} \iff s = \frac{-O_{pz}}{d_{pz}}. \tag{28.5}$$

[1] The hat symbol indicates normalized vectors.

Finally, we need to compute the texture coordinates (u_t, v_t):

$$u_t = \frac{-P_{px}}{w} + 0.5, \quad v_t = \frac{-P_{py}}{h} + 0.5. \tag{28.6}$$

For texture filtering, we need the partial derivatives of the texture coordinates, which can be computed easily when using ray differentials [5]. From the change in position $\partial\mathbf{P}/\partial\mathbf{x}$, we get

$$\frac{\partial u_t}{\partial x} = \frac{1}{w}\left(\frac{\partial \mathbf{P}}{\partial \mathbf{x}} \cdot \hat{\mathbf{t}}\right), \quad \frac{\partial v_t}{\partial x} = \frac{1}{h}\left(\frac{\partial \mathbf{P}}{\partial \mathbf{x}} \cdot \hat{\mathbf{b}}\right). \tag{28.7}$$

In order to use billboards with ray tracing, we need to put their bounding boxes into a ray tracing acceleration structure and write a custom intersection shader:

```
[shader("intersection")] void main() {
   Billboard b = billboards[InstanceID()][PrimitiveIndex()];
   float3x3 tbn = GetTangentSpace(WorldRayOrigin(), b); // Eq. 28.1
   // Transform ray to billboard coordinate space Eq. 28.3
   float3 d_p = mul(tbn, WorldRayDirection());
   float3 O_p = mul(tbn, WorldRayOrigin() - b.C);
   float s = -O_p.z / d_p.z; // Eq. 28.5
   if (s <= RayTMin()) return; // No intersection
   float2 P_p = O_p.xy + s * d_p.xy; // Eq. 28.4
   if(abs(P_p.x) > b.w * 0.5 || abs(P_p.y) > b.h * 0.5) return;
   BillboardIntersectionAttributes attr; // Struct with dummy
   ReportHit(s, 0, attr);
}
```

We observed that it was fastest to leave the intersection attributes empty and simply recompute the important variables in the any-hit or closest-hit shader.

28.2.2 REFLECTION AND REFRACTION ARTIFACTS

After hitting a reflective surface, a ray tracer will cast a new reflected ray from the intersection point. If we use the equations from the previous section to compute the billboard plane, we get different billboard planes for each reflected ray, because the ray origin is different for each ray (see Figure 28-2, left). Thus, the reflected billboard appears curved, as shown in Figure 28-3a.

To fix this problem, we need to choose the reflected ray origin as the common origin (Figure 28-2, right). Note that the intersection shader has access to only the ray description but not the ray payload. Therefore, the reflected ray should be cast from the common reflection origin and `Ray.TMin` has to be adjusted accordingly, as in Listing 28-1.

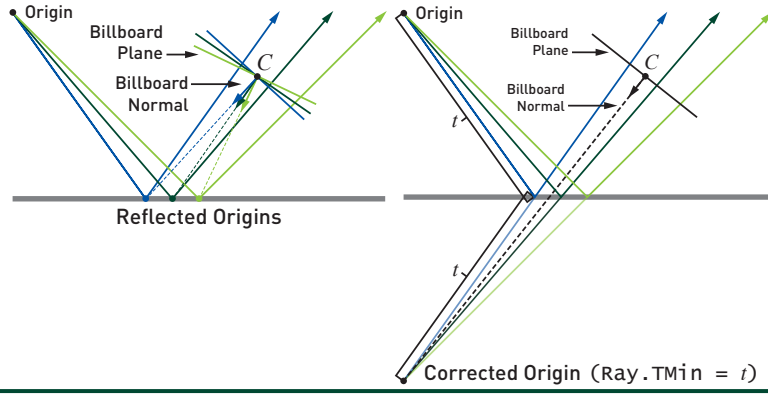


Figure 28-2. *Billboard planes after a reflection, using the reflected ray origin (left) and the corrected ray origin (right), where C is the billboard center.*

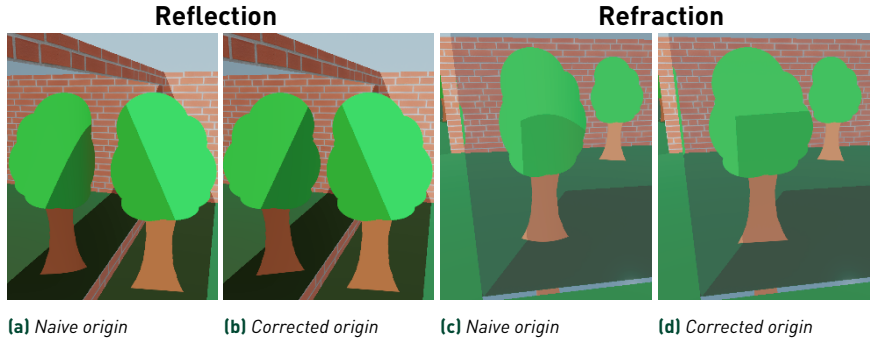
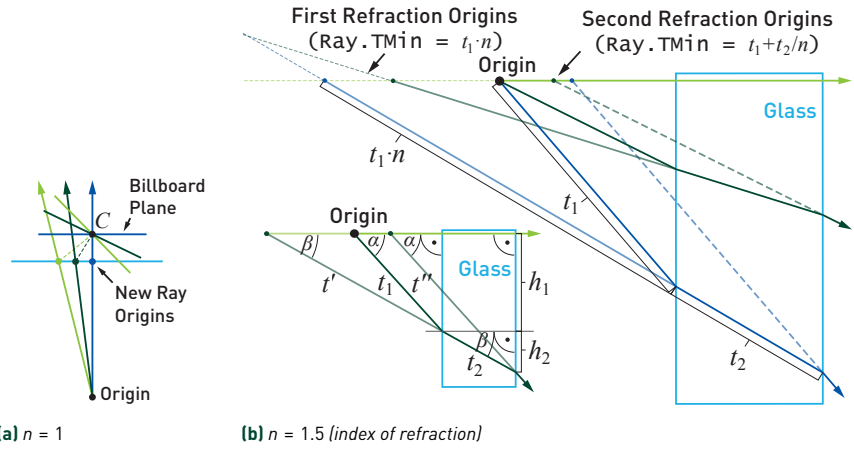


Figure 28-3. *Tree billboards without and with origin correction. The billboard is reflected in a mirror in the left pair of images and placed behind a glass pane in the right pair.*

Listing 28-1. *Corrected ray origin (reflections).*

```
float3 ipt = ray.Origin + ray.Direction * t;
newRay.Direction = reflect(ray.Direction, normal);
newRay.Origin = ipt - newRay.Direction * t;
newRay.TMin = t;
```

With this correction, we get a more plausible reflection, as shown in Figure 28-3b. Instead of computing the intersection point `ipt` as in Listing 28-1, we recommend using the technique presented by Wächter et al. [15] to avoid self-intersections due to floating-point issues. They describe a more precise computation for the new ray origin and also shift the origin in the direction of the surface normal.



(a) $n = 1$ **(b)** $n = 1.5$ *(index of refraction)*

Figure 28-4. *Billboard behavior after a refraction. (a) What happens to the billboard plane when new rays are cast from the new ray origins. (b) Refracted rays do not share a common origin anymore; this visualizes the origin correction technique.*

Unfortunately, a similar problem exists for refractions and transmissions. Transmissions without an index of refraction can be handled easily by keeping the same ray origin as before and only updating the `Ray.TMin` value instead of casting a new ray from the intersection point, as depicted in Figure 28-4a.

The situation becomes more complex for refractions. As shown in Figure 28-4b, the refracted rays do not share a common origin. We propose to project the individual ray origins onto the ray that is perpendicular to the intersection plane (light green ray). This way, all origins will be somewhat similar for refractions on planar surfaces. To project the origin of a refracted ray onto the perpendicular ray, we need to determine how far we need to move this origin in the negative refraction direction. This amount is depicted as t' for the first refraction and t'' for the second refraction. Both values can be derived with simple trigonometry and Snell's law (here, $\sin\beta = \frac{\sin\alpha}{n}$):

$$t' = \frac{h_1}{\sin\beta} = \frac{t_1 \sin\alpha}{\sin\beta} = \frac{t_1 n \sin\alpha}{\sin\alpha} = t_1 n,$$
$$t'' = \frac{h_1 + h_2}{\sin\alpha} = \frac{t_1 \sin\alpha + t_2 \sin\beta}{\sin\alpha} = \frac{t_1 \sin\alpha + \frac{t_2 \sin\alpha}{n}}{\sin\alpha} = t_1 + \frac{t_2}{n}. \quad (28.8)$$

The origin correction for refractions can then be expressed as in Listing 28-2.

We found that this approximation behaves well enough in practice to produce plausible-looking impostors after a refraction (see Figure 28-3d).

Listing 28-2. *Corrected ray origin (refractions).*

```
float eta = n1 / n2; // Ratio of refraction indices
float3 ipt = ray.Origin + ray.Direction * t;
newRay.Direction = refract(ray.Direction, normal, eta);
newRay.Origin = ipt - newRay.Direction * (t / eta);
newRay.TMin = t / eta;
```

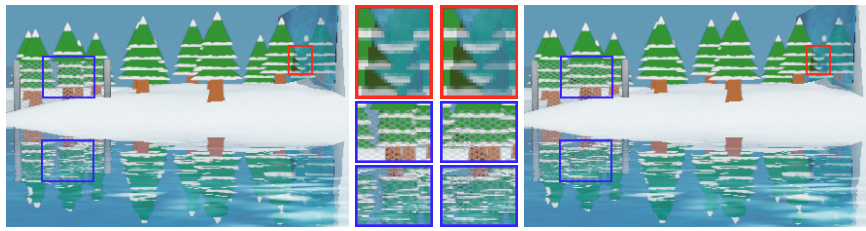


Figure 28-5. *Tree billboards reflected on ice and covered by an alpha-blended wired fence. The right image contains the origin correction for all trees. The blue boxes mark areas of interest for ray transmissions and the red boxes for ray reflections. Because the trees are aligned to the y-axis, they show no noticeable artifacts for reflections on the xz-plane.*

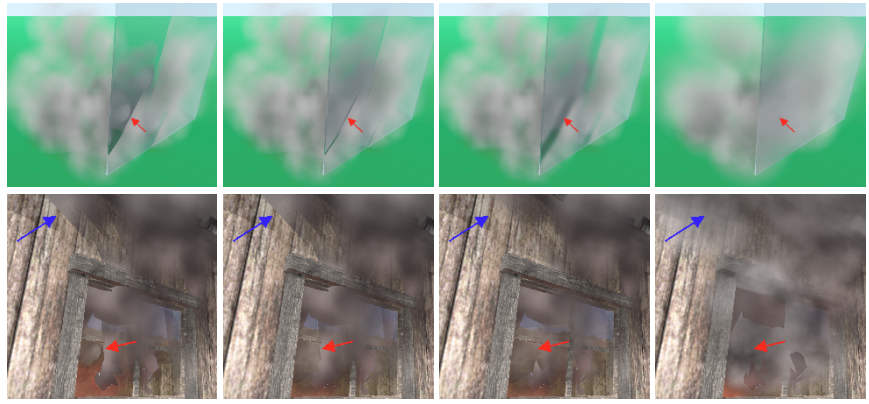
Note that impostors are only used if the object in question is far away. Figure 28-5 shows the impostors in a more realistic scenario. The artifacts can still be seen when looking closely at the highlighted areas, but they are not as noticeable as before. The ray origin correction is easy to implement and the only concern are potential precision problems for rays that travel long distances. We, however, did not encounter any precision related problems yet.

28.3 VOLUMETRIC EFFECTS

Volumetric effects are often simulated with a particle system [12]. However, instead of using thousands of particles to simulate realistic-looking volumetric effects, only a few dozens of the particles are simulated in a real-time context. Textured billboards are usually used to make up for the missing particles. The billboard textures generally occupy a spherical region of the billboard quad.

28.3.1 BILLBOARD PARTICLES

The particle billboards can be implemented similarly to the billboards from Section 28.2 but with a fixed radius $r = \frac{w}{2} = \frac{h}{2}$. Because the texture only occupies a spherical region, we can add another early-out condition to the intersection shader:



(a) *Naive origin* **(b)** *Corrected origin* **(c)** *Soft particles* **(d)** *Spherical part*

Figure 28-6. *Smoke rendered close to a glass pane (top) and a broken window (bottom). (The 3D model was created by [9].)*

```
1     if(P.x * P.x + P.y * P.y >= b.r * b.r) return;
```

Problems appear when particle billboards intersect other geometry: Figure 28-6a shows a naive implementation without the ray origin correction. The curved artifacts behind the glass pane can be fixed with the ray origin correction (Figure 28-6b). However, there are still some artifacts where the billboard cuts the glass pane or any opaque object like the ground plane or the wooden wall.

Soft particles [6] introduce an easy technique to mitigate those artifacts. When the position on the billboard plane is close to another (non-billboard) object, the opacity gets smoothly faded out. The original method uses a depth prepass to determine the distance to the opaque background per pixel. Unfortunately, this approach is not necessarily applicable in a ray tracing context. One way to implement it here is to trace a ray that determines the depth interval first:

```
// Determine depth range.
TraceRay(RAY_FLAG_SKIP_PROCEDURAL_PRIMITIVES, ray, payload);
ray.TMax = payload.rayTCurrent; // Set max range for particles.
const float rayTMin = ray.TMin; // Remember this value.
do { // Trace all particles.
   TraceRay(RAY_FLAG_SKIP_TRIANGLES, ray, particlePayload);
   alpha = CalcBillboardAlpha(particlePayload, rayTMin,ray.TMax);
   color += transmit*alpha * CalcBillboardColor(particlePayload);
   transmit *= (1.0 - alpha);
   ray.TMin = particlePayload.rayTCurrent; // Prepare for next particle.
} while(particlePayload.isValid);
color += transmit * CalcTriangleColor(payload);
```

Note that you can use the instance mask of `TraceRay()` as well to mask out the particle geometry. `CalcBillboardColor()` computes the color of the particle based on the texel color and light sources. `CalcBillboardAlpha()` determines the opacity based on the texel alpha and the current distance to `rayTMin` and `rayTMax`. It uses the `contrast(x)` function to smoothly fade out the opacity when being close to other scene geometry:

```
1  float contrast(float x) { // Smooth gradient for x in [0, 1].
2     if(x <= 0.5) return 0.5 * pow(2.0 * x, 2.0);
3     return 1.0 - 0.5 * pow(2.0 * (1.0 - x), 2.0);
4  }
5
6  float CalcBillboardAlpha(Payload p, float rayTMin, float rayTMax) {
7     float scale = 4 / p.r; // Scaling factor (r = billboard radius)
8     float outFade = contrast(saturate((rayTMax-p.rayTCurrent)*scale));
9     float inFade  = contrast(saturate((p.rayTCurrent-rayTMin)*scale));
10    return p.alpha * min(outFade, inFade); // Adjust particle opacity.
11 }
```

The final result is shown in Figure 28-6c. Unfortunately, there are still some artifacts where the billboards cuts the glass pane, but they are not as noticeable as before.

28.3.2 SPHERICAL PARTICLES

When cutting translucent objects, the artifacts can be solved completely if another interpretation of the billboards is used. Spherical billboards [14] interpret each billboard as a sphere with radius r instead of a viewer-oriented quadrilateral.

Here, the opacity α gets faded out based on the distance that the ray travels through the sphere Δs and the shortest distance to the sphere center l (Figure 28-7):

$$\alpha' = 1 - (1 - \alpha)^{\frac{\Delta s}{2r}\left(1-\frac{l}{r}\right)}. \tag{28.9}$$

The important variables can be calculated as follows:

$$\begin{aligned} t &= \widehat{\mathbf{d}} \cdot (C - O), \\ l &= |O - C + t\widehat{\mathbf{d}}|, \\ w &= \sqrt{r^2 - l^2}, \\ \Delta s &= \min(Z_{\text{far}}, t + w) - \max(Z_{\text{near}}, t - w). \end{aligned} \tag{28.10}$$

Next is the normal and texture coordinate calculation. Instead of projecting the intersection onto a 2D plane to obtain texture coordinates, we took this

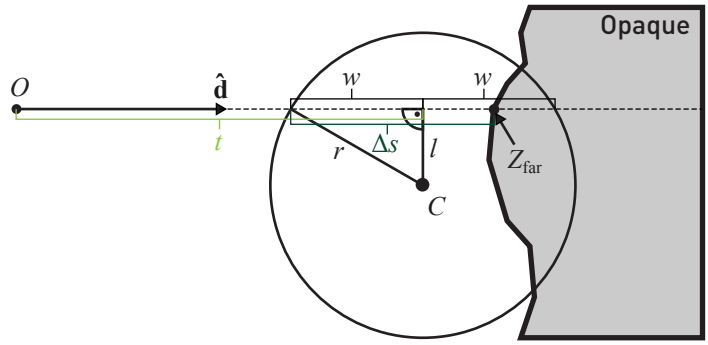


Figure 28-7. *A spherical billboard, where Z_{far} describes the distance to the first opaque object.*



(a) *2D particle* **(b)** *Lat.-long. particle texture* **(c)** *Ray tracing spheres*

Figure 28-8. *Particle textures and possible self-intersections.*

opportunity to get rid of the last major artifact of traditional billboards. Because normal billboards are viewer-oriented quads, they rotate when the viewer moves. This means that the textures on top of the billboards rotate as well (Figure 28-8a). To avoid this artifact, we decided to use the polar coordinates (θ, ϕ) of the sphere in combination with a latitude-longitude texture (see Figure 28-8b):

$$\begin{aligned} \hat{\mathbf{n}} &= \frac{O + (t - w)\hat{\mathbf{d}} - C}{r}, \\ \theta &= \arctan(n_z, n_x), \quad \phi = \arccos(n_y), \\ u_t &= \frac{\theta}{2\pi} + \xi_i, \quad v_t = \frac{\phi}{\pi}. \end{aligned} \tag{28.11}$$

Note that we use only the fractional part of u_t in which we added a random offset ξ_i that gives each billboard a different texture rotation. The texture derivatives are more complicated to compute for the spherical billboards. The book *Physically Based Rendering* [10] explains how to evaluate ray differentials for any given surface. In order to apply that procedure, we have to calculate the change in surface coordinates P when moving in UV space. For our

spheres, these derivatives evaluate to

$$\frac{\partial \mathbf{P}}{\partial \mathbf{u_t}} = 2\pi r \begin{pmatrix} -\sin\phi\sin\theta \\ 0 \\ \sin\phi\cos\theta \end{pmatrix}, \quad \frac{\partial \mathbf{P}}{\partial \mathbf{v_t}} = \pi r \begin{pmatrix} \cos\phi\cos\theta \\ -\sin\phi \\ \cos\phi sin\theta \end{pmatrix}. \tag{28.12}$$

As these derivatives span the tangent plane of the surface at the point P, there will be a linear combination that results in $\partial P/\partial x$. The coefficients for this combination are the change in the texture coordinates. The previously described derivatives have three components, but we only want to solve for two coefficients. Therefore, we omit the component that has the highest contribution to the normal, which usually guarantees that the resulting 2×2 system is uniquely solvable. We therefore define the helper function

$$\tau : \begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{cases} (y,z)^\mathsf{T} & \text{if } |n_x| = \arg\max\|n\|, \\ (x,z)^\mathsf{T} & \text{if } |n_y| = \arg\max\|n\|, \\ (x,y)^\mathsf{T} & \text{if } |n_z| = \arg\max\|n\|, \end{cases} \tag{28.13}$$

which is used for the following 2×2 system:

$$\tau\left(\frac{\partial \mathbf{P}}{\partial \mathbf{x}}\right) = \left[\tau\left(\frac{\partial \mathbf{P}}{\partial \mathbf{u_t}}\right) \; \tau\left(\frac{\partial \mathbf{P}}{\partial \mathbf{v_t}}\right)\right] \begin{pmatrix} \frac{\partial u_t}{\partial x} \\ \frac{\partial v_t}{\partial x} \end{pmatrix}, \tag{28.14}$$

which can be solved for $\partial u_t/\partial x$ and $\partial v_t/\partial x$.

The intersection shader reports the foremost intersection with the sphere:

```
[shader("intersection")] void main() {
    Billboard b = billboards[InstanceID()][PrimitiveIndex()];
    float t = dot(WorldRayDirection(), b.C - WorldRayOrigin());
    float l = length(WorldRayOrigin()-b.C+t*WorldRayDirection());
    if(l >= r) return; // Ray misses the sphere.
    float w = sqrt(b.r * b.r - l * l);
    float rayTIn = max(RayTMin(), t - w);
    float ds = min(RayTCurrent() /*TMax*/, t + w) - rayTIn;
    if(ds <= 0.0) return; // Invalid depth range

    BillboardIntersectionAttributes attribs;
    ReportHit(rayTIn, 0, attribs);
}
```

The ray tracing kernel is similar to that in Section 28.3.1 because we need to know the depth interval as well. However, as we no longer report hits on a 2D plane but inside a sphere, we need to handle special cases to avoid potential self-intersections.

Take a look at Figure 28-8c. After obtaining the foremost intersection from Ray 1 with sphere S_1, tracing a ray from that intersection point would result in

a self-intersection with S_1 based on our intersection shader (`RayTCurrent = RayTMin`). If we ignore intersections from rays that start inside a sphere, we would get the intersection S_2 next. However, this approach would result in no intersections for the second ray (Ray 2). Furthermore, we noticed that a sorted composition with the `OVER` operator [11] is wrong when working with deep (volumetric) samples. Duff [4] describes how to blend deep samples. A list with all deep samples needs to be maintained to obtain the correct solution. This, however, is too expensive in a real-time context, especially if the maximum number of samples is unknown.

We propose a workaround for real-time applications. One way to query all intersections in a ray interval is to use any-hit ray tracing instead of closest-hit ray tracing. The downside of this approach is that the any-hit shaders are not necessarily invoked in front-to-back order. However, we do not have to worry that intersections are reported multiple times if we specify the `GEOMETRY_FLAG_NO_DUPLICATE_ANYHIT_INVOCATION` when initializing the geometry. Thus, we could use an *order-independent transparency* (OIT) technique in the any-hit shader to determine the color and transmittance.

We decided to use weighted-blended OIT [8] because we are working with many particles of similar color and transmittance. In weighted-blended OIT, the particles are combined with the following (commutative) formula:

$$\mathbf{c}_f = \frac{\sum_{i=1}^{n} \mathbf{c}_i w(z_i, \alpha_i)}{\sum_{i=1}^{n} \alpha_i w(z_i, \alpha_i)} \left(1 - t_f\right), \quad t_f = \prod_{i=1}^{n} (1 - \alpha_i), \tag{28.15}$$

$$w(z, \alpha) = \alpha \operatorname{clamp}\left((10(1 - 0.99 z_{\text{nonlinear}})^3), 0.01, 3000\right), \tag{28.16}$$

where $\mathbf{c}_i$ and α_i are the pre-multiplied color and opacity of each particle, c_f is the color of all particles combined, and t_f is the remaining transmittance. Note that $z_{\text{nonlinear}}$ refers to the nonlinear transformation of z (see [8, Equation 11]). The shader implementation for spherical billboard ray tracing is:

```
void trace(inout float color, inout float transmit, RayDesc ray){
   BillboardPayload p;
   p.rayTMax = ray.TMax; // Cannot be accessed via intrinsics
   p.colorSum = p.alphaSum = 0.0;
   p.transmit = 1.0;
   TraceRay(RAY_FLAG_SKIP_TRIANGLES | RAY_FLAG_SKIP_CLOSEST_HIT_SHADER, ray,
         p);

   if(p.transmit < 1.0) // Eq. 28.15
      color += transmit * (p.colorSum / p.alphaSum)
         * (1.0 - p.transmit);
   transmit *= p.transmit;
}
```

```
[shader("anyhit")]
void anyHit(BillboardPayload p, BillboardIntersectionAttributes){
    float4 c = GetBillboardColor(p.rayTMax);
    // Do color accumulation as in weighted-blended OIT.
    float weight = w(RayTCurrent(), c.a); // Eq. 28.16
    p.colorSum += c.rgb * weight;
    p.alphaSum += c.a * weight;
    p.transmit *= (1.0 - c.a);
    IgnoreHit(); // Keep any-hit traversal going.
}
```

Notice how the `trace()` function only traces a single ray to capture all fragments in the specified ray interval. The final result can be seen in Figure 28-6d. Finally, no more artifacts are visible when intersecting other scene geometry.

28.4 EVALUATION

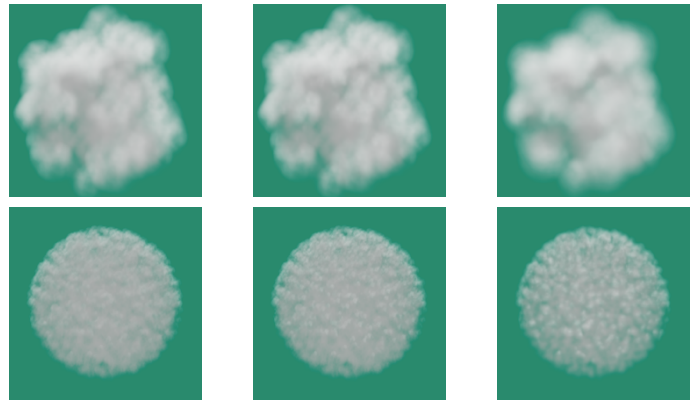
28.4.1 PERFORMANCE

Billboards implemented with procedural primitives (axis-aligned bounding boxes) are significantly slower than their baked triangle representation. This is mainly because the built-in triangle intersection test is faster than a custom intersection shader call. We measured that procedural billboards result in up to 40% longer rendering times [2, p. 43]. We recommend creating an additional ray tracing acceleration structure that contains the camera-oriented billboards in their triangle representation. This additional acceleration structure should be used for primary rays, or rays that were affected by little refraction and no reflection.

Table 28-1 shows that our spherical particles are faster than our soft particles. This is due to any-hit ray tracing being used for spherical particles instead of closest-hit ray tracing. For a fair comparison, we included times for a soft particles variant that uses weighted-blended OIT with any-hit ray tracing. Here, the spherical particles are slower because they are more complex to compute. Note that our soft particles become faster when the opacity of each particle increases, which allows termination of the ray early in closest-hit ray tracing.

28.4.2 LIMITATIONS

For planar surfaces, our ray origin correction produces correct billboards after reflections and also gives a good approximation for refractions. However, it does not always produce the expected results for curved surfaces (see Figure 28-9). Here, the billboard refraction can be corrected by using the



Particle Count	Soft Particles		Soft Particles OIT		Spherical Particles	
	No Shad.	Shadow	No Shad.	Shadow	No Shad.	Shadow
0	0.6 ms	0.6 ms	0.8 ms	0.8 ms	0.8 ms	0.8 ms
100	3.0 ms	9.2 ms	1.4 ms	7.0 ms	1.4 ms	7.6 ms
1000	8.7 ms	22.4 ms	2.4 ms	16.7 ms	2.8 ms	18.6 ms
10,000	26.6 ms	55.2 ms	5.9 ms	46.2 ms	7.2 ms	50.7 ms

Table 28-1. *Smoke cloud rendered with ray traced soft particles, soft particles using weighted-blended OIT, and spherical particles. The table lists the rendering times for 0 to 10,000 particles without and with ray traced shadows (top figures are made with 100 particles, bottom figures with 10,000 particles). All times are in milliseconds and recorded on an NVIDIA RTX 2080 TI graphics card.*

ray origin of the primary ray for the billboard-normal computation. However, because the primary ray origin is not necessarily a point on the refracted ray (as shown in Figure 28-4b), this information cannot be encoded in the ray itself. As the ray payload is inaccessible from the intersection shader, the only place to store such information is a potentially slow global memory buffer. However, because the artifacts in Figure 28-9 are only noticeable because large billboards are placed very close to a reflective or refractive object, a more precise correction does not appear to be necessary in typical situations.

28.5 CONCLUSION

We presented an easy way to adapt different kinds of billboards for ray tracing and mitigated possible artifacts.

The origin correction presented in Section 28.2 handles artifacts that occur when billboards are placed close to a reflective or translucent object.

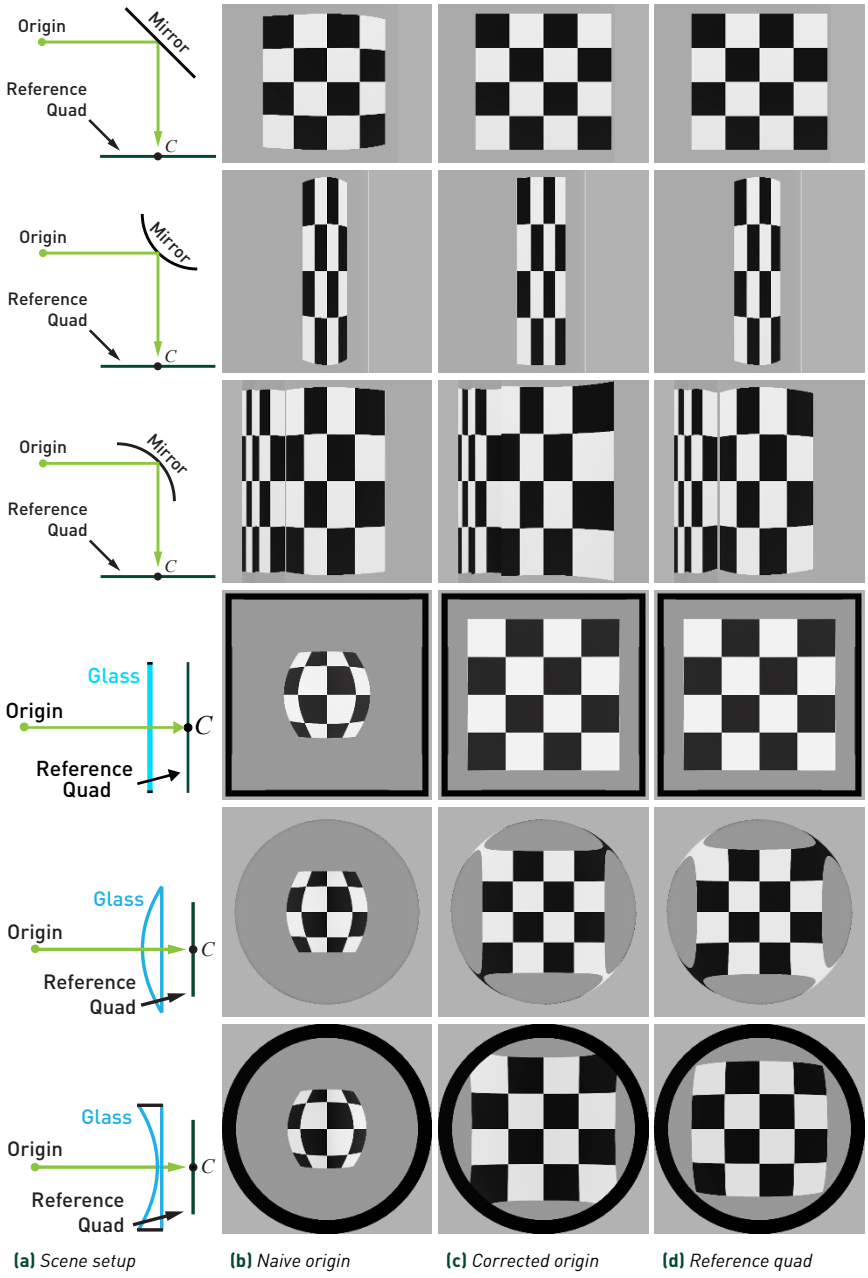


Figure 28-9. *(a) The scene setup for the current row. (b) The scene rendered for a billboard with a naive ray origin. (c) Our origin correction. (d) Rendering with the reference quad.*

Although this correction can be negligible for impostors, as they are usually drawn at a very small size, the origin correction is of high significance when rendering billboard particles behind translucent objects (see the circular artifacts in Figure 28-6a).

The soft particles technique prevents hard cuts with other scene geometry. However, due to the contrast function, there are still some artifacts when a particle intersects with translucent objects (see Figure 28-6c).

Finally, we presented the adaptation of spherical billboards for ray tracing. Even though we had to improvise to blend multiple particles, the final result looks fine and does not contain artifacts when billboards cut through translucent objects (see Figure 28-6d). In the future, we would like to replace weighted-blended OIT with another technique that can handle a broader range of materials.

The prototype was written with Falcor [1]. Code samples are available at the book's source code website and the full project with all scenes can be found at https://github.com/kopaka1822/Falcor/tree/billboards.

ACKNOWLEDGMENTS

We would like to thank our section editor Cem Yuksel for the great guidance and his suggestion to use the index of refraction to improve the ray origin correction.

REFERENCES

[1] Benty, N., Yao, K.-H., Clarberg, P., Chen, L., Kallweit, S., Foley, T., Oakes, M., Lavelle, C., and Wyman, C. The Falcor rendering framework. https://github.com/NVIDIAGameWorks/Falcor, 2020. Accessed August 2020.

[2] Brüll, F. *Fast Transparency and Billboard Ray Tracing with RTX Hardware*. Master thesis, Clausthal University of Technology, Oct. 2020. DOI: 10.13140/RG.2.2.14692.19842.

[3] Décoret, X., Durand, F., Sillion, F. X., and Dorsey, J. Billboard clouds for extreme model simplification. *ACM Transactions on Graphics*, 22(3):689–696, July 2003. DOI: 10.1145/882262.882326.

[4] Duff, T. Deep compositing using Lie algebras. *ACM Transactions on Graphics*, 36(4):120a:1–120a:12, June 2017. DOI: 10.1145/3072959.3023386.

[5] Igehy, H. Tracing ray differentials. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 179–186, 1999. DOI: 10.1145/311535.311555.

[6] Lorach, T. Soft Particles. Technical report, NVIDIA, 2007. https://developer.download.nvidia.com/whitepapers/2007/SDK10/SoftParticles_hi.pdf.

[7] Maciel, P. W. C. and Shirley, P. Visual navigation of large environments using textured clusters. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 95–102, 1995. DOI: 10.1145/199404.199420.

[8] McGuire, M. and Bavoil, L. Weighted blended order-independent transparency. *Journal of Computer Graphics Techniques*, 2(2):122–141, 2013. http://jcgt.org/published/0002/02/09/.

[9] OliverMH. Woodcutter's cabin. http://www.blendswap.com/blends/view/86149, October 2, 2016.

[10] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., 3rd edition, 2016. http://www.pbr-book.org/3ed-2018/Texture/Sampling_and_Antialiasing.html#FindingtheTextureSamplingRate.

[11] Porter, T. and Duff, T. Compositing digital images. *ACM SIGGRAPH Computer Graphics*, 18(3):253–259, Jan. 1984. DOI: 10.1145/964965.808606.

[12] Reeves, W. T. Particle systems—A technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, Apr. 1983. DOI: 10.1145/357318.357320.

[13] Ryan, B. Octahedral impostors. https://www.shaderbits.com/blog/octahedral-impostors, March 18, 2018.

[14] Umenhoffer, T., Szirmay-Kalos, L., and Szijártó, G. Spherical billboards and their application to rendering explosions. In *Proceedings of Graphics Interface 2006*, pages 57–63, 2006. DOI: 10.5555/1143079.1143089.

[15] Wächter, C. and Binder, N. A fast and robust method for avoiding self-intersection. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 77–85. Apress, 2019. DOI: 10.1007/978-1-4842-4427-2_6.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 29

HYBRID RAY TRACED AND IMAGE-SPACE REFRACTIONS

Daniel Parhizgar and Marcus Svensson
Avalanche Studios

ABSTRACT

With the recent release of inline ray tracing, it is possible to traverse and register ray hits inside a pixel shader. This allows for convenient implementation of ray tracing into past rasterization algorithms and possibly improve on their limitations. One such rasterization algorithm is image-space refractions, which are used for approximating refractions in real time. Image-space refractions are limited in that they can be highly inaccurate when refracting through concave objects containing more than two surface layers. This chapter combines image-space refractions with inline ray tracing into a hybrid method that overcomes such limitations and generates more realistic refractions. The results are closer to fully ray traced images, but using fewer rays. By replacing some rays with image-space refractions, our hybrid method improves rendering speed as compared to pure ray traced refractions.

29.1 INTRODUCTION

The use of image-space methods to simulate refractions through two surfaces in real time was first presented by Wyman [6]. This approach of computing refractions in a rasterizer could produce results close to ray tracing, but was limited to closed surfaces containing two depth layers. Since then, Wyman [7] proposed a method to achieve proper ray intersections with geometries located in the background, and Davis and Wyman [2] presented a solution for the total internal reflection phenomenon.

Krüger et al. [4] suggested using *depth peeling* [3] to extract multiple layers into depth buffers. This allowed handling refractions through more than two depth layers. However, this approach was not extensively evaluated in the context of image-space refractions. Also, their work includes an optimization for computing a ray intersection using a lookup in the next depth layer to

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_29

reduce the number of texture accesses. Unfortunately, when using depth peeling for refraction, this simplification does not work properly on certain concave objects such as a donut [1].

With the introduction of inline ray tracing to DirectX Raytracing (DXR) Tier 1.1, it is possible to start the ray tracing process for small tasks inside the pixel shader. This allows implementing ray tracing into any existing rasterization-based algorithms, such as image-space refractions, and potentially eliminates some of their limitations.

This chapter presents a hybrid refraction method that combines image-space refractions with inline ray tracing for fast and realistic refraction computation through a refractive object. This hybrid method provides a solution for refractions through more than two depth layers, a feature not supported by pure image-space refractions. The refractions are computed inside a pixel shader along with inline ray tracing. In comparison to pure ray traced refractions, our hybrid method achieves faster rendering with visually close results by reducing the number of rays required for resolving the refractions (Figure 29-1).

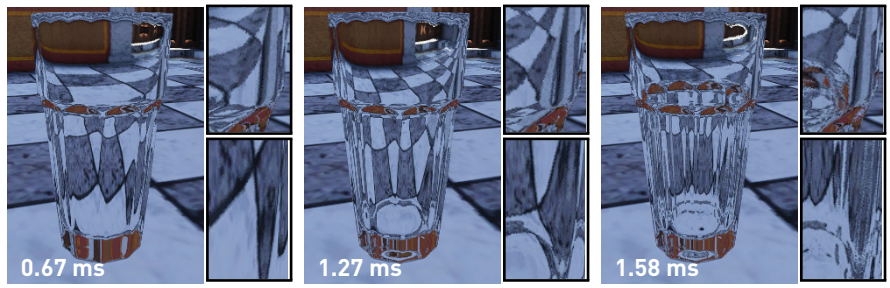
29.2 IMAGE-SPACE REFRACTIONS

The original image-space refraction method [6] uses a prepass for rendering backfacing surface depth and normals into textures. This is accomplished by simply reversing the depth test. These textures are then sampled during rendering to retrieve information about the refractive object's backfacing surface layer and perform refractions there.

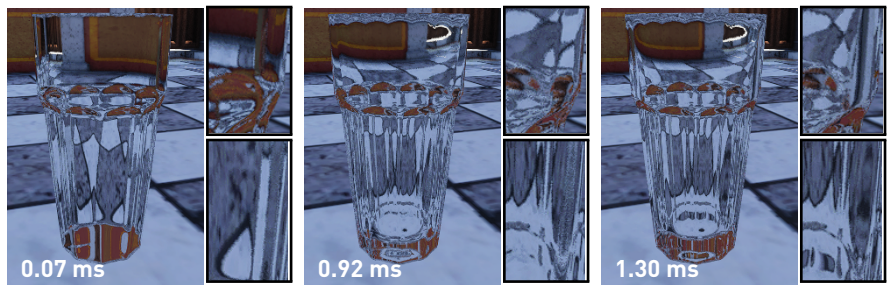
In a pixel shader, one has access to the fragment position P_1, the surface normal $\mathbf{n}_1$, and the incoming view direction $\mathbf{v}$. With these vectors and the ratio of the refractive indices η, the refraction direction $\mathbf{t}_1$ for the first refraction event can be calculated using the HLSL function `refract()`. This refraction direction is then used for approximating the hit point P_2 on the backfacing surface using

$$P_2 \approx P_1 + \tilde{d}\,\mathbf{t}_1, \tag{29.1}$$

where $\tilde{d}$ is the approximated distance to the backfacing surface. Here, $\tilde{d}$ can be taken as the difference between the current depth at position P_1 and the depth value in the backfacing depth texture at the same pixel. Though $\tilde{d}$ is an inaccurate travel distance for a refraction ray inside the object, it provides a measure of the object's thickness. Therefore, it can be an acceptable approximation for the travel distance.



(a) *Ray tracing: one ray* **(b)** *Ray tracing: two rays* **(c)** *Ray tracing: three rays*



(d) *Image-space reference* **(e)** *Hybrid: two post-rays* **(f)** *Hybrid: two pre-rays*

Figure 29-1. *Comparison of (a–c) ray traced refractions, (d) image-space refractions, and (e–f) our hybrid refractions with post- and pre-rays. Using ray traced refractions, three rays are needed in this case to resolve the refractions of the glass. Our hybrid method with two pre-rays produces visually similar results to ray tracing with three rays.*

After P_2 is approximated, it is projected onto the backfacing normal texture, where a normal $\mathbf{n}_2$ is sampled. Using the `refract()` function again, this time with values $\mathbf{n}_2$ and $\mathbf{t}_1$, produces the second refraction direction $\mathbf{t}_2$ at the backfacing surface layer of the object. Then, $\mathbf{t}_2$ is used for sampling an environment texture, and the final color value is retrieved. This way, refraction through the two depth layers of an object is approximated without tracing rays (see Figure 29-2).

29.3 HYBRID REFRACTIONS

The idea with hybrid refractions is to add ray tracing to image-space refractions. There are two options here: the first is to perform ray tracing before the image-space refractions, and the second is to perform ray tracing after the image-space refractions. We refer to these two options as *pre-rays* and *post-rays*, respectively. There are benefits as well as disadvantages with both options.

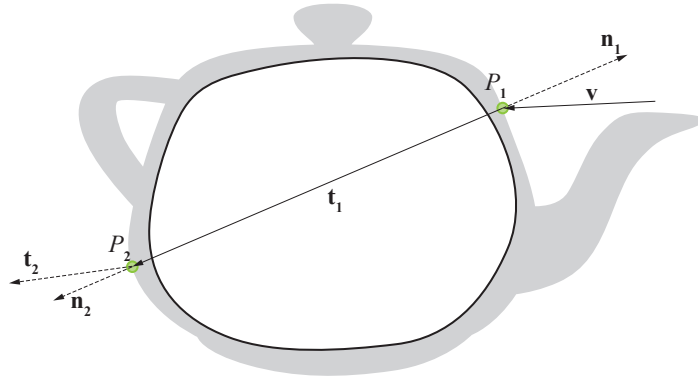


Figure 29-2. *In a pixel shader, the incident vector* $\mathbf{v}$*, the normal* $\mathbf{n}_1$ *and the position* P_1 *are already known for each pixel. Using these, the transmitted or refracted vector* $\mathbf{t}_1$ *can be computed. The unknown variables, position* P_2 *and normal* $\mathbf{n}_2$*, are retrieved by projecting* $\mathbf{t}_1$ *onto the backfacing surface textures. Once these are calculated, a second refraction can be performed [6].*

29.3.1 HYBRID REFRACTIONS WITH PRE-RAYS

With the pre-rays option, ray traced refractions are added to the image-space method after the first refraction direction $\mathbf{t}_1$ has been calculated using the HLSL function `refract`$(\mathbf{v}_1, \mathbf{n}_1, \eta)$. Then, we begin computing a user-defined fixed number of refraction events using ray tracing. After the ray tracing part is done, the final refraction event is performed using the image-space approach. The final ray hit position P_i is projected onto the backfacing depth texture. The last refraction distance d is approximated as the difference between the depth value at P_i and the backfacing depth texture value that corresponds to the pixel position of P_i.

The image-space refraction at the end accounts for a single refraction event. However, this final refraction takes place at the backfacing surface of the object, skipping any remaining intermediate levels. Therefore, the resulting refractions include the backfacing surface detail, even when the number of refraction events we handle would not be sufficient to reach the backfacing surface with pure ray tracing.

29.3.2 HYBRID REFRACTIONS WITH POST-RAYS

The idea of the post-rays method is to perform the image-space refractions first and do the ray tracing later. In this case, the image-space approach is used for computing the refractions through the *second* surface layer of the

object, instead of the last layer (i.e., the backfacing surface of the object) used by the pre-rays option. Therefore, the normal and depth textures store the second surface layer, instead of the backfacing surface (i.e., the last) layer. In order to render the second layer's depths and normals into the textures, we simply use frontface culling (instead of reversing the depth test, which is done with pre-rays).

29.3.3 PRE-RAYS VS. POST-RAYS

With the post-rays option, any inaccuracies in image-space refractions occur earlier. Therefore, such errors are more likely to lead to visual artifacts in the final image. Because the second layer depth and normals are stored in textures, the backfacing surface detail could be missing in the resulting refractions, if the ray traced refractions cannot reach the backfacing surface. This happens when too few post-rays are used. Another potential artifact with the post-rays option can occur when the image-space projection is supposed to have ended on the backfacing surface, but at that position the second layer normal texture is sampled from, which causes a faulty refraction. The significance of these inaccuracies depends on the object shape and the view angle.

Both options in our hybrid refractions can handle up to a fixed number of refraction events per pixel. If certain pixels require fewer refraction events, the pre-rays option ends up skipping the image-space refraction computation and the post-rays option skips computing some rays. Therefore, the post-rays option can provide additional reductions in ray traversal computation for some pixels, thereby achieving faster render times.

The saving in speed can make the post-rays option favorable in certain situations. When computing refractions in most graphics applications, as long as there is some form of plausible refractive distortion, it can be sufficient to produce believable results. Therefore, the inaccuracies of the post-rays option may not be an important problem, as it is difficult to guess how physically accurate refractions are supposed to look in many cases. Also, it can still handle refractions through multiple surfaces, providing improved fidelity over pure image-space refractions.

29.4 IMPLEMENTATION

The hybrid refractions method starts by performing the first refraction event, which is always handled without ray tracing:

```
float3 HybridRefractPreRays() {
  float3 V = FragmentWorldPos - CameraPos;
  float3 P1 = FragmentWorldPos;
  float3 N1 = SurfaceNormal;
  // Ratio between indices of refraction
  float refr_ratio = index_air/index_glass;
  float3 T1 = refract(normalize(V), normalize(N1), refr_ratio);
```

After the first refraction, using the pre-rays option, ray traced refractions with a fixed recursion depth begin. This is where inline ray tracing is used. Documentation and code examples for inline ray tracing can be found on the DXR Functional Spec GitHub page [5].

```
  RayQuery<RAY_FLAG_CULL_NON_OPAQUE |
           RAY_FLAG_SKIP_PROCEDURAL_PRIMITIVES> rq;

  RayDesc ray;
  ray.Origin = P1;
  ray.Direction = T1;
  ray.TMin = 0.001;
  ray.TMax = 100;

  for (int i = 0; i <= max_depth; ++i)
  {
    rq.TraceRayInline(AccelerationStructure, 0, ~0, ray);
    rq.Proceed();

    if (rq.CommittedStatus() == COMMITTED_TRIANGLE_HIT)
    {
      RayInfo ray_info =
            RayInstancesBuffer[rq.CommittedInstanceID()];

      if (Refractive surface was hit)
      {
        float3 normal = CalculateSurfaceNormal(
                          ray_info,
                          rq.CommittedTriangleBarycentrics(),
                          rq.CommittedPrimitiveIndex(),
                          rq.CommittedObjectToWorld3x4()
                        );

        float3 hit_position = ray.Origin + normalize(ray.Direction)*rq.
            CommittedRayT();
        if(Final backfacing surface was hit)
        {
          float3 dir = refract(normalize(ray.Direction), normalize(-normal),
                index_glass/index_air);
          dir = CheckForTotalInternalReflection();
          float3 color = ProjectToColorTexture(hit_position + dir);
          return color;
        }

        if (dot(normalize(normal), normalize(ray.Direction)) < 0)
        {
          refr_ratio = index_air/index_glass;
        }
```

```
49          else
50          {
51            // A backfacing surface has been hit;
52            // normal and refr_ratio have to adjust.
53            normal *= -1;
54            refr_ratio = index_glass/index_air;
55          }
56
57
58          RayDesc newRay;
59          newRay.Direction = refract(normalize(ray.Direction), normalize(
                 normal), refr_ratio);
60          // Check for total internal reflection.
61          if (abs(newRay.Direction) < 1e-6)
62          {
63            newRay.Direction = reflect(normalize(ray.Direction), normal);
64          }
65
66          newRay.Origin = hit_position;
67          newRay.TMin = 0.001;
68          newRay.TMax = 100;
69
70          ray = newRay;
71          continue;
72        }
73        else
74        {
75          // A non-refractive surface has been hit.
76          // Sample the resulting color from the environment map.
77          float3 hit_position = ray.Origin + ray.Direction*rq.CommittedRayT();
78          return ProjectToColorTexture(hit_position);
79        }
80      }
81      else if (rq.CommittedStatus() == COMMITTED_NOTHING)
82      {
83        // Miss shader
84        return ProjectToColorTexture(ray.Origin);
85      }
86    }
```

After the ray tracing part is done, the distance $\tilde{d}$ is computed and the current ray position is projected onto the backfacing normal texture, so that the final refraction can be computed. On line 93 in the following code, an if-statement tests whether the last ray ended on the backfacing surface of the object. This is performed by simply comparing the magnitude of the distance $\tilde{d}$ to zero (or a small epsilon). When the last ray is on the backfacing surface, it means that pre-rays have completely resolved the refraction and there is no need for an image-space refraction step.

```
87    float uv = ConvertToUV(ray.Origin);
88    float3 back_pos = BackFacingPosTexture.Sample(uv);
89    float d = distance(back_pos, ray.Origin); // Compute distance d.
90    float3 T2 = ray.Direction;
91    float3 P2 = ray.Origin;
```

```
92
93    if(abs(d) > 1e-6)
94    {
95      P2 = ray.Origin + d * ray.Direction;
96      // Project and sample N2.
97      float3 N2 = ProjectToBackFacingNormalTexture(P2);
98
99      refr_ratio = index_glass/index_air; // Swap indices of refraction.
100     T2 = refract(normalize(ray.Direction), normalize(-N2), refr_ratio);
101
102     // Check for total internal reflection.
103     if (abs(T2) < 1e-6)
104     {
105       T2 = reflect(normalize(ray.Direction), normalize(-N2));
106     }
107   }
108
109   float3 P3 = P2 + T2;
110   float color = ProjectToColorTexture(P3);
111   return color;
112 }
```

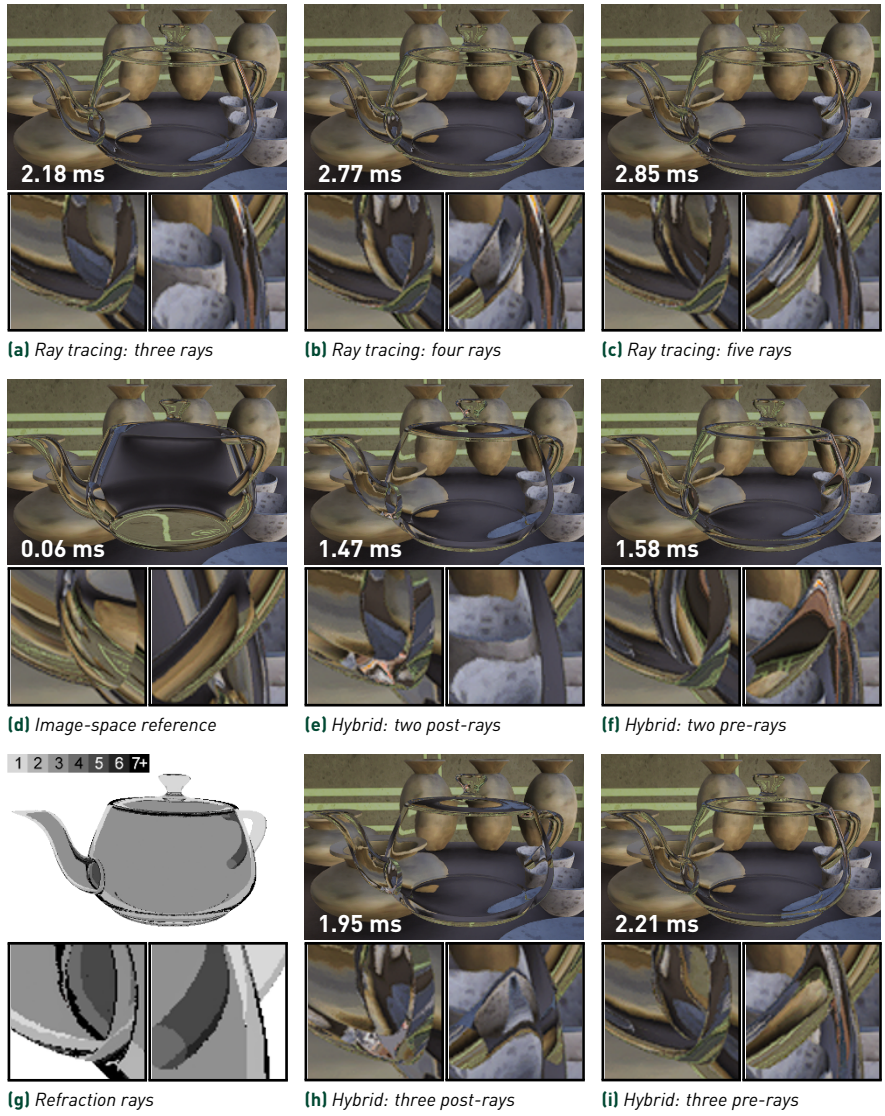
The implementation of the post-rays option is similar. The main difference is that it starts with image-space refraction, instead of ray tracing. Also, there is no way to check if the image-space refraction ends up on the backfacing surface, so at least one ray is always computed with the post-rays option.

```
1 void HybridRefractPostRays()
2 {
3     // Perform image-space refraction first.
4     // Store resulting position and direction in a structure.
5     RayDesc ray = ImageSpaceRefract();
6     // Use that structure as the starting point for ray traced refractions.
7     RayTraceRefract(ray);
8 }
```

29.5 RESULTS

We present comparisons of different refraction computation methods with two different models in Figures 29-1 and 29-3. All timings are computed on a computer with an Intel i7-8700K CPU with 3.7 GHz, 32.0 GB RAM, and an NVIDIA GeForce RTX 2070 SUPER graphics card. The reported timings are the mean values of the five executions, and they include the precomputation time for image-space refractions. The screen resolution was set to 1920×1080 pixels.

As can be seen in Figure 29-1, ray tracing can compute proper refractions, but it requires a certain number of rays per pixel to fully resolve them. Using only one or two rays per pixel, refractions through the backside of the glass are missing (Figures 29-1a and 29-1b). Fixing the refractions through the backfacing side requires at least three rays (Figure 29-1c).



(a) *Ray tracing: three rays* **(b)** *Ray tracing: four rays* **(c)** *Ray tracing: five rays*

(d) *Image-space reference* **(e)** *Hybrid: two post-rays* **(f)** *Hybrid: two pre-rays*

(g) *Refraction rays* **(h)** *Hybrid: three post-rays* **(i)** *Hybrid: three pre-rays*

Figure 29-3. *Comparison of (a–c) ray traced refractions, (d) image-space refractions, and (e,f,h,i) our hybrid refractions with post- and pre-rays, along with (g) a visualization of the number of rays needed to fully resolve the refractions of each pixel with ray tracing.*

Our hybrid refractions with two pre-rays (Figure 29-1f) produces visually indistinguishable results from ray tracing with three rays per pixel (Figure 29-1c), but with a reduction in render time. Our hybrid refractions with two post-rays reduces the render time even further (Figure 29-1e), but there are small visual differences near the edges of the glass.

Refractions through a modified Utah teapot model with a thin surface and hollow interior are presented in Figure 29-3. Again, the results show faster render times using hybrid refractions with the post-rays option, as compared to the pre-rays option. The post-rays option, however, has missing details or significant deviations from the pure ray traced refractions on parts of the teapot: on the lid, near the silhouettes, and where the spout joins the body (Figures 29-3e and 29-3h). Hybrid refractions with pre-rays always includes refraction details due to the backfacing surface, even when too few rays are used, such as the part of the handle that joins the body visible in Figure 29-3f. As expected, using more rays (Figure 29-3i) produces closer results to pure ray tracing.

Though both hybrid options were always faster than pure ray traced refractions, the speed improvement with the pre-rays option is reduced as the fixed pre-ray count is increased. This is because, as the pre-ray count increases, fewer pixels perform image-space refraction, as the refractions are resolved prior to the image-space refraction step. The post-rays option, however, always performs image-space refractions, regardless of the fixed post-ray count. Therefore, it maintains its speed improvement with increasing post-ray count.

The image-space refractions method requires only a fraction of the render time, as compared to all other methods, because it does not use any ray tracing. However, it cannot handle refractions through multiple layers correctly (Figures 29-1d and 29-3d). The refractions on the glass and teapot are completely different than the ones produced by all other methods.

29.6 CONCLUSION

In this chapter, we presented how inline ray traced refractions can be integrated with image-space refractions in order to create two variations of a hybrid method. Our hybrid refractions method uses ray tracing either after (post-rays) or before (pre-rays) the image-space refractions. Both variations have their uses: the post-rays option is faster, whereas the pre-rays option is more realistic. The visual results of the pre-rays option come close to refractions computed with pure ray tracing using more rays per pixel. Also, our hybrid method addresses an important limitation of the image-space refractions approach by properly handling multi-layer refractive objects.

ACKNOWLEDGMENTS

A special thanks to Cem Yuksel for all the feedback and improvements suggested to the implementations and for the Utah teapot model, to Marcus Svensson for supervising and providing much needed help throughout the project, to Johan Pauli for importing the 3D models to the APEX engine, and to Avalanche Studios for providing all the equipment. This work was made in Avalanche Studios' APEX engine.

REFERENCES

[1] Chi, M.-T., Wang, S.-H., and Hu, C.-C. Real-time multiple refraction by using image-space technology. *Journal of Information Science and Engineering*, 32:625–641, May 2016.

[2] Davis, S. T. and Wyman, C. Interactive refractions with total internal reflection. In *Proceedings of Graphics Interface 2007*, pages 185–190, 2007. DOI: 10.1145/1268517.1268548.

[3] Everitt, C. Interactive order-independent transparency. Technical report, NVIDIA, 2001. https://www.nvidia.com/en-us/drivers/Interactive-Order-Transparency/.

[4] Krüger, J., Bürger, K., and Westermann, R. Interactive screen-space accurate photon tracing on GPUs. In *17th Eurographics Symposium on Rendering*, pages 319–329, 2006. DOI: 10.2312/EGWR/EGSR06/319-329.

[5] Microsoft, Patel, A., and Hargreaves, S. DirectX Raytracing (DXR) functional spec. https://github.com/microsoft/DirectX-Specs/blob/2f0ec17689d51d8dfc36c1eff690a1d87110b3fc/d3d/Raytracing.md#inline-raytracing, 2020. Accessed July 7, 2020.

[6] Wyman, C. An approximate image-space approach for interactive refraction. *ACM Transactions on Graphics*, 24(3):1050–1053, July 2005. DOI: 10.1145/1073204.1073310.

[7] Wyman, C. Interactive image-space refraction of nearby geometry. In *Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, pages 205–211, 2005. DOI: 10.1145/1101389.1101431.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 30

REAL-TIME RAY TRACED CAUSTICS

Xueqing Yang and Yaobin Ouyang
NVIDIA

ABSTRACT

We present two real-time ray tracing techniques for rendering caustic effects. One focuses on the caustics around metallic and transparent surfaces after multiple ray bounces, which is built upon an adaptive photon scattering approach and can depict accurate details in the caustic patterns. The other is specialized for the caustics cast by water surfaces after one-bounce reflection or refraction, which is an enhancement of the algorithm of caustics mapping, highly fluidic in sync with the water ripples, and able to cover large scene areas. Both techniques are low cost for high frame rate usages, fully interactive with dynamic surroundings, and ready-to-use with no data formating or preprocessing requirements.

30.1 INTRODUCTION

Caustics are commonly seen phenomenon in scenes containing water, metallic, or transparent surfaces. However, in most of today's real-time renderers, they are either ignored or roughly handled using tricks like decal textures. Although objects casting caustics may only occupy a small portion of the screen in most cases, the delicate optical patterns are very challenging to simulate with a limited time budget. Fortunately, the arrival of GPU ray tracing brings out the possibility of performing photon mapping [6]—the most efficient technique for simulating caustics so far—in real time to accurately rendering these effects.

Noticeably, in the book *Ray Tracing Gems*, Hyuk Kim [7] proposed a simple scheme to execute photon mapping in real time: tracing photons through the scene, blending them directly onto a screen-space buffer, and then applying a spatial denoiser to obtain the final patterns. Albeit easy to implement, the method uses a fixed resolution for photon emission with uniform distribution, which limits its application for large-scale scenes, and the blend-denoise

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_30



Figure 30-1. *Screenshots of real-time ray traced caustics. Top: the classic "POV-Ray glasses" (courtesy of Gilles Tran). The caustics and glass meshes are ray-traced up to 12-bounce refraction and reflection. Bottom: the undersea water caustic effect from the game JX3 HD Remake developed by Kingsoft Seasun Studio.*

process is prone to exhibit either blurry or noisy results. In the same book, Holger Gruen [4] showed an improved caustics mapping algorithm for underwater caustics, which traced photons from a rasterized water mesh and reconstructed the lighting in screen space. Although being able to eliminate most artifacts in some earlier attempts of rendering underwater caustics, the algorithm still cannot produce sharp but noise-free caustic patterns.

To simulate high-quality caustics in real time by utilizing GPU ray tracing (see Figure 30-1), we present two techniques in this chapter:

1. *Adaptive Anisotropic Photon Scattering (AAPS):* The AAPS technique presented in Section 30.2 is for generating caustics around high-polished metallic and transparent objects. It facilitates traditional

forward photon tracing with photon differentials [5, 10], an anisotropic approach to obtain finer scattering quality and higher efficiency. In addition, inspired by [1, 3, 11], the technique handles photon emission adaptively to generate highly detailed caustic patterns in local regions and to maintain temporal stability. A soft caustic algorithm for area light sources is also provided.

2. *Ray-Guided Water Caustics (RGWC):* The RGWC technique presented in Section 30.3 is for creating caustics above and under water surfaces. It is a continuation of Gruen's work [4] and improves it over several aspects: intensifying the optical details in water caustic by applying photon difference or procedural meshes; supporting most light types and their relevant properties, including textured area lights; being able to cover vast scene regions through cascaded caustics maps; and being flexible on performance-quality trade-off with user-controlled bias-variance preference.

30.2 ADAPTIVE ANISOTROPIC PHOTON SCATTERING

The AAPS method simulates caustics by revamping some techniques from photon splatting, a category of light propagation methods that is a variation of classic photon mapping. In the previous GPU-based method [8], photons are shot and drawn as isotropic particles. AAPS modifies these particles to project elliptical footprints by evaluating photon differentials during hit-bounce time, and then invoking the rasterization pipeline to draw them against the scene depth with additive blending. Such an anisotropic setup can significantly save the bandwidth and bring superior details.

Besides the anisotropic photon setup, our algorithm has two additional novelties:

- A negative feedback loop to distribute photons adaptively into the important areas, i.e., regions close to the camera or parts of the screen where the outcome exhibits temporal instability.
- "Soft caustics" cast by area light sources, which is done by modifying photon differential information at the emission stage.

The algorithm maintains the following buffers:

- *Task buffers:* A set of buffers for guiding the photon emission in the current frame, including the quadtree buffer and the light ID buffer (Section 30.2.1).

- *Photon buffer:* A structured buffer recording photon data, including hit position, photon footprint, and intensity.
- *Feedback buffers:* A couple sets of textures in the light space for tracking feedback information, including the average screen-space area of photon footprints, caustics variance, and ray density (Section 30.2.3).
- *Caustics buffer:* A render target for splatting photons in the screen space.

The workflow is executed in four steps (see Figure 30-2):

1. Emit photons according to the task buffers and trace them through the scene. For any photon hitting an opaque surface, create a record in the photon buffer and add its footprint area to the feedback buffers.
2. Perform photon scattering (splatting) on the caustics buffer: each photon in the photon buffer is drawn as an elliptical footprint against the scene depth. The shape and intensity of the footprint are calculated from photon differentials and the surface normal, respectively.
3. Apply the caustics buffer to the scene, which is usually done by accessing the scene attributes in the G-buffer and performing a deferred lighting pass.
4. Combine the feedback buffers of the previous frame and the current frame to generate the task buffers for the next frame.

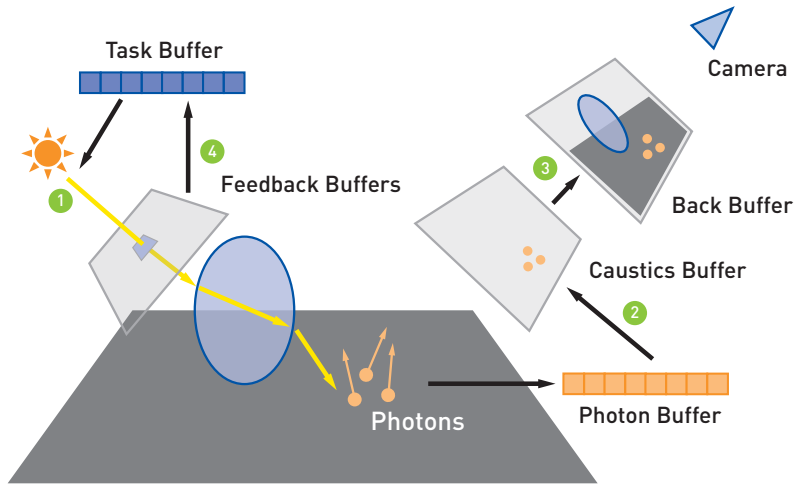


Figure 30-2. *The AAPS workflow. The numbered circles relate to the numbered list in the text.*

In the rest of this section, we first elaborate these four steps in detail (Sections 30.2.1–30.2.3) and describe two approximating methods for simulating dispersion (Section 30.2.4) and soft caustics (Section 30.2.5), respectively. Then, we show our results and the performance tests in typical applications and give some guidance on optimizing performance (Section 30.2.6). Finally, we discuss the algorithm's limitations (Section 30.2.7) and present extended usages (Section 30.2.8).

30.2.1 PHOTON TRACING

To implement adaptive photon emission, we maintain a 2D texture called the *ray density texture* to guide photon distribution, in which all light sources' light-space views are tiled together to track the per-pixel ray count that should be traced (Figure 30-3). The texture is then expanded into a sequence of global ray IDs for photon emission that are placed in a 1D buffer called the *quadtree buffer*, in which a quadtree is constructed for accelerating queries and is updated every frame. An additional 2D texture called the *light ID buffer* is also created for light source lookup.

During photon tracing, each ray generation shader thread is dispatched to trace one ray from one of the light sources. The light source ID and the ray's location in the ray density texture are obtained by querying the quadtree buffer, and the ray's origin or direction is determined by the location mapped to one of the light spaces. Figure 30-4 shows an example of this searching process: First, the shader thread searches for the ray's location, starting from the top of the tree and traversing down to the leaf node whose ID range

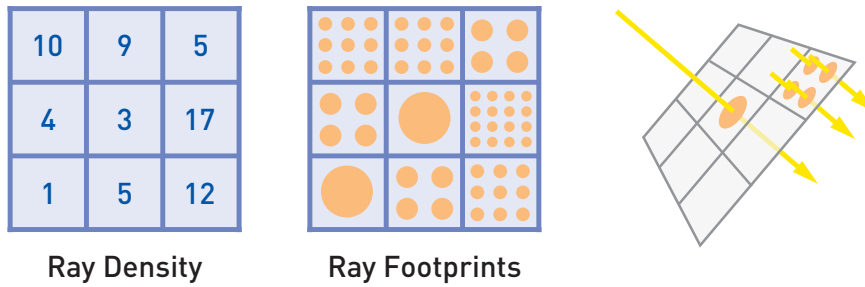


Figure 30-3. *The ray density texture in light space. Left: the per-pixel ray count in the ray density texture. Middle: visualized ray distribution and initial photon footprint size derived from ray density. For each pixel, we set the ray count to the nearest square number less than the number of rays. Right: shooting rays according to the samples' locations in the texture.*

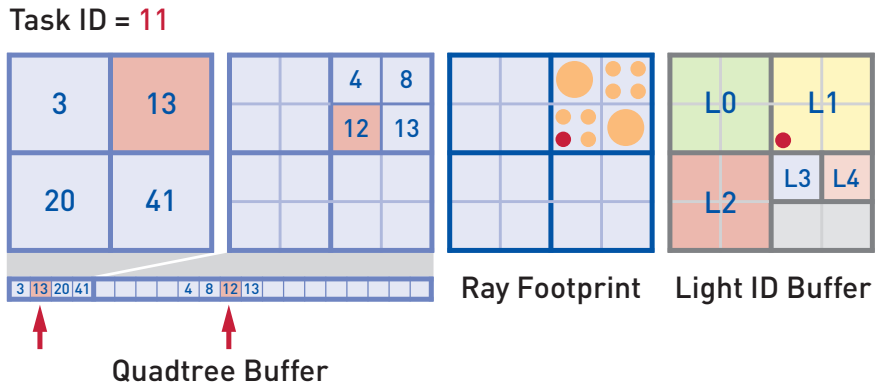


Figure 30-4. *An example of the ray query process: In the quadtree buffer, each quadtree node's ID equals the highest ray ID inside its subtrees, and the ray count in the subtree equals its ID subtracted by its previous sibling's. For a thread with task ID 11, the ray generation shader starts searching from four subtrees 3, 13, 20, and 41 containing 4, 10, 7, and 21 rays, respectively; then, it traverses into subtree 13, which contains four subtrees 4, 8, 12, and 13; finally, it traverses into subtree 12 to find the matching ray ID 11. The 2D location of ray ID 11 is then used for retrieving the light source data from the light ID buffer and generating the UV parameters to sample the light source.*

contains the thread ID. Then, it uses the location to look in the light ID buffer and find out which light source to sample, and it calculates the UV parameters and transforms the location into the light space. Finally, the thread computes the initial size of the photon footprint from the ray density at that position and then sets up a ray to trace the photon. The quadtree query code looks as follows:

```
// taskId is calculated from thread ID.
// sampleIdx is the task offset inside current quadtree node.
uint2 pixelPos = 0;
uint sampleIdx = taskId;
uint4 value = RayCountQuadTree[0];

// Discard threads that don't have a task.
if (taskId >= value.w)
    return;

// Traverse quadtree.
for (int mip = 1; mip <= MipmapDepth; mip++)
{
    pixelPos <<= 1;
    if (sampleIdx >= value.b)
    {
        pixelPos += int2(1, 1);
        sampleIdx -= value.b;
    }
```

```
    else if (sampleIdx >= value.g)
    {
        pixelPos += int2(0, 1);
        sampleIdx -= value.g;
    }
    else if (sampleIdx >= value.r)
    {
        pixelPos += int2(1, 0);
        sampleIdx -= value.r;
    }
    // Calculate linear index based on mipmap level and pixel position.
    int nodeOffset = GetTextureOffset(pixelPos, mip);
    value = RayCountQuadTree[nodeOffset];
}

// lightInfo = {light ID, range, anchor point X, anchor point Y}
uint4 lightInfo = LightIDBuffer.Load(int3(pixelPos, 0));
// Get pixel size from ray density texture.
float2 pixelSize; uint2 lightAtlasCoord;
GetRaySample(pixelPos, sampleIdx, lightAtlasCoord, pixelSize);
// Calculate light UV for ray configuration and delta UV for footprint.
float2 lightUV = (lightAtlasCoord - lightInfo.zw) / lightInfo.y;
float2 deltaUV = pixelSize / lightInfo.y;
```

Tracking the photon footprints during the ray tracing is done by estimating photon differentials [10]. Supposing a ray shot from a light has two positional parameters $\mathbf{p} = \mathbf{p}(u, v)$ in the case of a directional light source, or two directional parameters $\mathbf{d} = \mathbf{d}(u, v)$ in the case of a point light, the photon's hit position $\mathbf{p}'$ after the ray tracing is determined by all parameters $\mathbf{p}' = \mathbf{p}'(u, v)$. The algorithm generates two small perturbations for each ray, updates them using the chain rule when the photon hits a surface, and uses the perturbations of photon position $\Delta\mathbf{p}'$ to determine the new size of the photon footprint:

$$\Delta\mathbf{p}' = \frac{\partial\mathbf{p}'}{\partial u}\Delta u + \frac{\partial\mathbf{p}'}{\partial v}\Delta v. \tag{30.1}$$

The photon differentials are evaluated in closest-hit shaders. Once reaching a visible, opaque, and rough surface, the photon's attributes are recorded in the photon buffer, including the hit position, intensity, incident direction, and final footprint, which is determined by the partial derivatives $\partial\mathbf{p}'/\partial u$ and $\partial\mathbf{p}'/\partial v$ at the last hit point.

30.2.2 PHOTON SCATTERING

In the photon scattering step, we construct the photon footprints as quadrilaterals based on their differentials and then draw them into the caustics buffer using additive blending. During the rendering, each footprint's lighting result is computed by the photon's intensity, incident angle, and

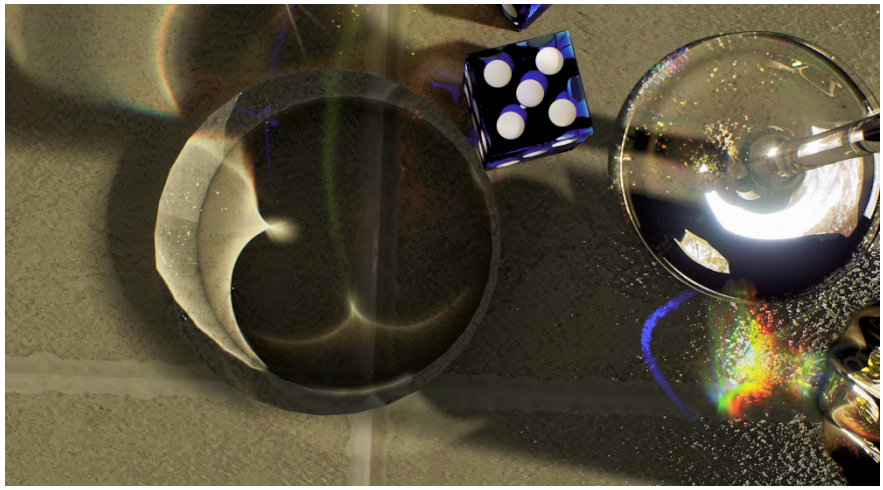


Figure 30-5. *Mesh caustics cast on a glossy surface. A GGX shading pass is performed for each photon footprint.*

surface attributes at the hit point retrieved from the G-buffer. Figure 30-5 shows an example in which the caustics are cast on a glossy surface, where the incident angle and intensity from the photon, the local geometry, and the material information from the G-buffer are collected to perform a GGX shading pass.

After photon scattering, a compute shader back-projects the current screen pixels to the previous frame to calculate the caustics variance between the two frames, and it stores the result in the alpha channel of caustics buffer. The variance values are used for ray density calculation, which is a part of the feedback mechanism (described in the next section).

30.2.3 FEEDBACK BUFFERS

The AAPS technique features a mechanism of a negative feedback loop to determine the photon distribution adaptively. At its core, a couple of textures, together called *feedback buffers*, are placed in the light spaces and updated in each frame to evaluate the spatiotemporal importance of the traced photons (Figure 30-6).

The *projected area texture* contains the average screen-space area of the photon footprints. At the end of photon tracing, each photon's final footprint is calculated and its area in the screen space is added to the light-space location in this texture where the photon was emitted.

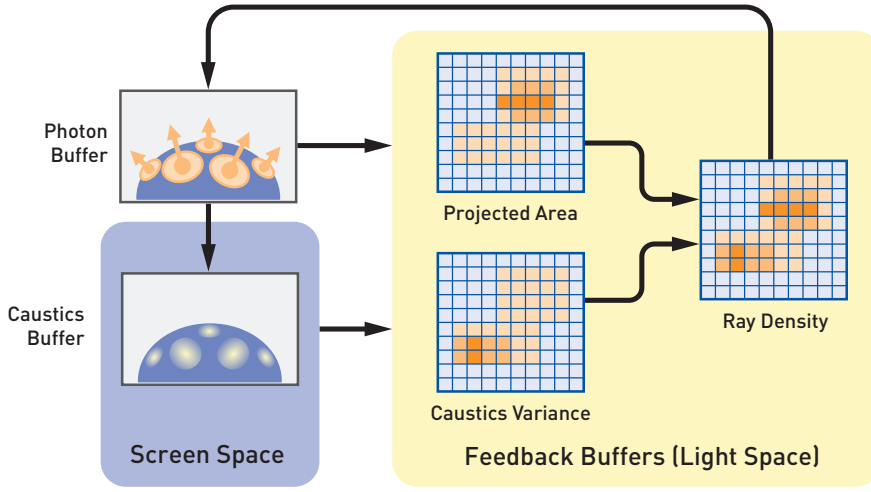


Figure 30-6. *The mechanism of feedback buffers. Top left: at the end of the photon tracing stage, each photon footprint's screen-space area is calculated and added to the projected area texture in light space. Bottom left: the caustics variance between the current and previous frames is stored in the caustics buffer. At the end of the photon tracing stage, each photon samples variance from the caustics buffer and accumulates it into the caustics variance texture in light space. Right: the projected area and caustics variance are combined to generate the ray density.*

The *caustics variance texture* contains the average color variance of the screen-space pixels covered by the photons. At the end of photon scattering, each photon collects the variance value from the caustics buffer's alpha channel at its footprint's center and writes the value into the pixel of this texture where the photon was emitted. This looks as follows:

```
1  // At the end of photon tracing, calculate screen-space coordinates and
       footprint area.
2  float3 screenCoord;
3  float pixelArea = GetPhotonScreenArea(photon.position, photon.dPdx, photon.
       dPdy, screenCoord);
4  // Read variance value from caustics buffer.
5  float variance = GetVariance(screenCoord);
6  // Write feedback buffers; lightAtlasCoord is the 2D coordinate in light
       space.
7  uint dummy;
8  InterlockedAdd(ProjectedArea[lightAtlasCoord], uint(pixelArea), dummy);
9  InterlockedAdd(Variance[lightAtlasCoord], uint(variance), dummy);
10 InterlockedAdd(PhotonCount[lightAtlasCoord], 1, dummy);
```

The *ray density texture* is computed for the next frame by combining the two previous textures to compute a suggested ray density, which is used as a

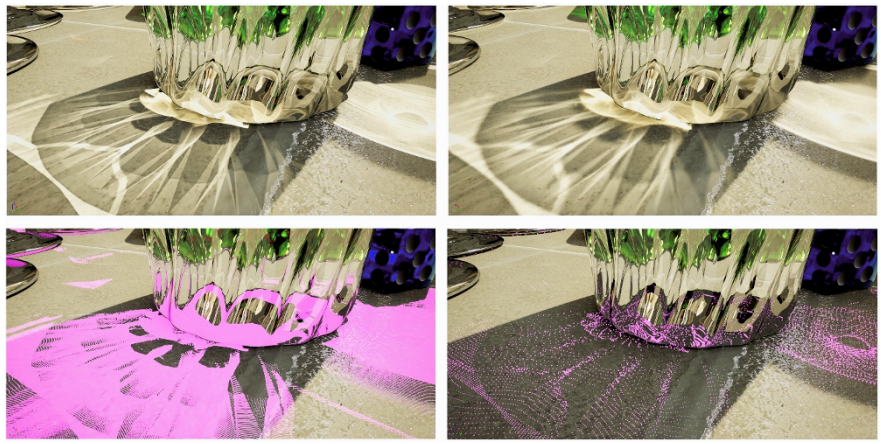


Figure 30-7. *Comparison of target projected area of photon footprint set to 20 (upper left) and 80 (upper right). The second row shows the point visualization of the photons. Detailed caustics patterns are well captured without apparent noise.*

guide for the per-pixel ray count:

$$d' = d\frac{a}{a_t} + vg, \tag{30.2}$$

where d' is the suggested ray density, d is the previous ray density, a is the average screen-space projected area, a_t is the target projected area, v is the caustics variance, and g is the variance gain. In order to create sharper details, we can set a_t to a smaller value to restrict photon size (Figure 30-7 shows a comparison between two a_t values); and to suppress temporal flickering, we can set g to a higher value.

Applying d' directly for subsequent usage may cause the small local features to be unstable if the change is too steep. To avoid this, we filter d' by blending with the neighboring pixels' current ray density:

$$d_{\text{final}} = w_t d' + (1 - w_t)\frac{\sum_i w_i d_i}{\sum_i w_i}, \tag{30.3}$$

where d_{final} is the final ray density, d' is the suggested ray density, the d_i are the current ray densities of the pixel and its neighbors, and w_t and w_i are temporal and spatial weights, respectively, which should be set between 0 and 1. A higher value of w_t enables faster updates but less stable results.

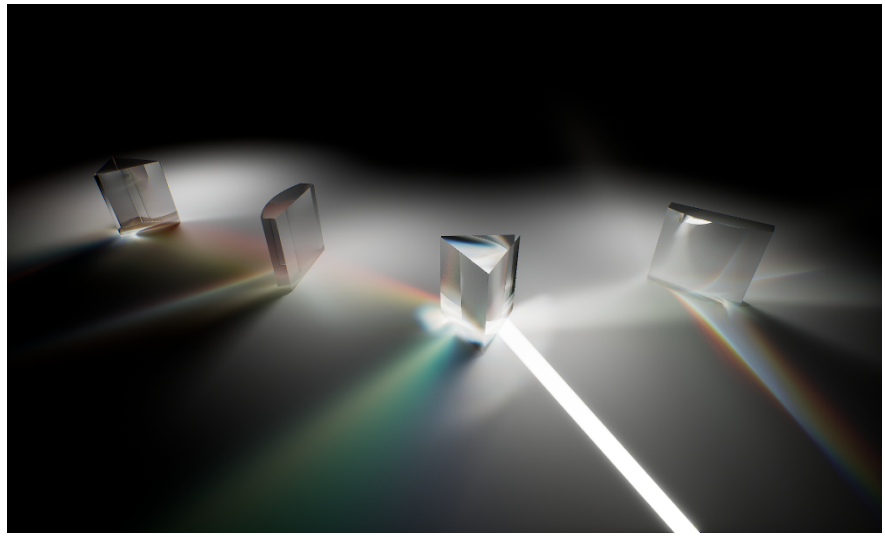


Figure 30-8. *Light dispersion though prisms.*

30.2.4 DISPERSION

Accurately simulating light dispersion (as shown in Figure 30-8) requires computing spectral ray differentials, as done by Elek et al. [2], plus the emission spectrum of light sources and the absorption spectrum of materials. To avoid such complicated input data and huge computational cost, we employ a perturbation-based approach instead.

First, at each refraction point, an index of refraction (IOR) perturbation is calculated based on the thread ID:

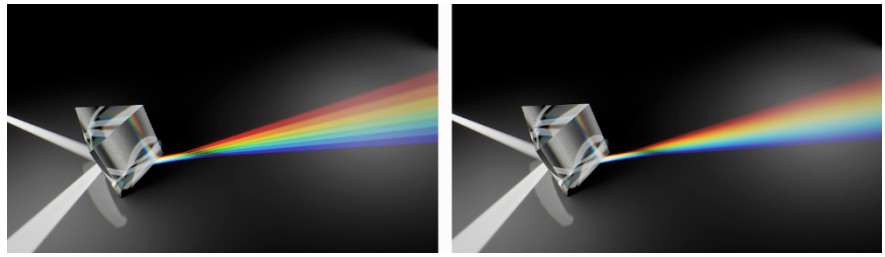
$$\Delta i = 2\frac{t \bmod s_d}{s_d - 1} - 1, \tag{30.4}$$

where Δi is the IOR perturbation in the range [–1, 1], t is the thread ID, and s_d is the number of separated monochromatic colors. In Figure 30-9a, s_d is set to 7, thus the white light is split into seven monochromatic colors. Then, Δi is applied to modify the IOR and generate the refraction ray.

Next, at the end of the photon tracing stage, Δi is used for calculating the modulation color. The RGB triplet of modulation weights can be computed by

$$C_f = saturate\left(-\Delta i, w_g(1 - |\Delta i|), \Delta i\right), \tag{30.5}$$

where C_f is the modulation color in RGB channels and w_g is the weight factor



(a) *Seven color bands* **(b)** *Jittered IOR perturbation*

Figure 30-9. *Monochromatic color separation: (a) The white light is split into seven color bands. (b) Continuous color separation with jittering is applied.*

for the green channel, which ensures that all modulation colors can be combined into grayscale colors:

$$w_g = \frac{s_d + 1}{2s_d - 2}. \tag{30.6}$$

Finally, each photon's color is multiplied by its modulation color C_f. The dispersion effect now looks like a series of colorful bands. To make the color distribution smoother over the spectrum, the IOR perturbation Δi can be jittered. Figure 30-9b shows the results with jittering.

30.2.5 SOFT CAUSTICS

AAPS can simulate soft caustics cast by area light sources. Unlike directional or point light source, an area light source can emit photons with independently varied position and direction. Thus, we need to formulate a proper method on estimating photon differentials based on the four perturbations.

Suppose that a photon emitted from an area light has two positional parameters $\mathbf{p} = \mathbf{p}(u, v)$ and two directional parameters $\mathbf{d} = \mathbf{d}(p, q)$. The final hit point of the photon $\mathbf{p}'$ is determined by all parameters $\mathbf{p}' = \mathbf{p}'(u, v, p, q)$. Adding either a positional or a directional perturbation to the ray's origin will raise a shift to the hit point (Figure 30-10). Our solution for area light sources is to treat all four perturbations $\Delta u, \Delta v, \Delta p, \Delta q$ as independent random variables obeying standard normal distribution and the photon footprint as the significant area of resulting probability distribution. The perturbations of the

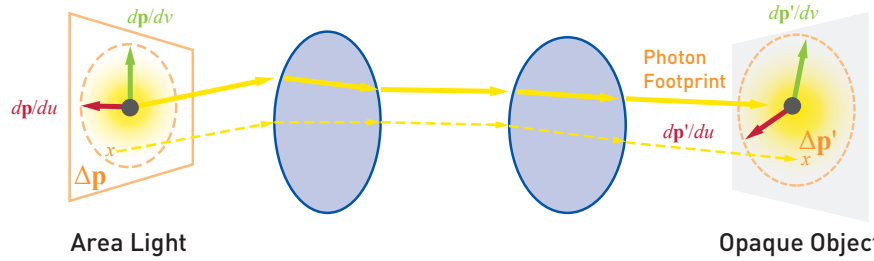


Figure 30-10. *Adding a positional perturbation* $\Delta\mathbf{p}$ *on the ray's origin will raise a positional perturbation* $\Delta\mathbf{p}'$ *on the hit point.*

photon position are $\Delta\mathbf{p}' = \Delta\mathbf{p}'_\mathbf{p} + \Delta\mathbf{p}'_\mathbf{d}$, in which

$$\Delta\mathbf{p}'_\mathbf{p} = \frac{\partial\mathbf{p}'}{\partial u}\Delta u + \frac{\partial\mathbf{p}'}{\partial v}\Delta v, \tag{30.7}$$

$$\Delta\mathbf{p}'_\mathbf{d} = \frac{\partial\mathbf{p}'}{\partial p}\Delta p + \frac{\partial\mathbf{p}'}{\partial q}\Delta q. \tag{30.8}$$

Here, $\Delta\mathbf{p}'_\mathbf{p}$ and $\Delta\mathbf{p}'_\mathbf{d}$ are 2D photon perturbation vectors in the local *xy*-coordinate frame of the photon, raised by positional and directional perturbations, respectively. Because all perturbation inputs are normally distributed, both $\Delta\mathbf{p}'_\mathbf{p}$ and $\Delta\mathbf{p}'_\mathbf{d}$ obey normal distribution: $\Delta\mathbf{p}'_\mathbf{p} \sim N(\mathbf{0}, \mathbf{C_p})$ $\Delta\mathbf{p}'_\mathbf{d} \sim N(\mathbf{0}, \mathbf{C_d})$ in which

$$\mathbf{C_p} = \begin{pmatrix} \frac{\partial\mathbf{p}'}{\partial u} & \frac{\partial\mathbf{p}'}{\partial v} \end{pmatrix} \begin{pmatrix} \frac{\partial\mathbf{p}'}{\partial u} & \frac{\partial\mathbf{p}'}{\partial v} \end{pmatrix}^T, \tag{30.9}$$

$$\mathbf{C_d} = \begin{pmatrix} \frac{\partial\mathbf{p}'}{\partial p} & \frac{\partial\mathbf{p}'}{\partial q} \end{pmatrix} \begin{pmatrix} \frac{\partial\mathbf{p}'}{\partial p} & \frac{\partial\mathbf{p}'}{\partial q} \end{pmatrix}^T. \tag{30.10}$$

Because $\Delta\mathbf{p}'_\mathbf{p}$ and $\Delta\mathbf{p}'_\mathbf{d}$ are independent, the probability distribution of $\Delta\mathbf{p}'$ is the convolution of the probability distributions of $\Delta\mathbf{p}'_\mathbf{p}$ and $\Delta\mathbf{p}'_\mathbf{d}$. Note that the convolution of two normal distributions is still a normal distribution, with the mean value and the covariance matrix being the sum of the two respectively:

$$\Delta\mathbf{p}' \sim N(\mathbf{0}, \mathbf{C}), \tag{30.11}$$

where $\mathbf{C} = \mathbf{C_p} + \mathbf{C_d}$. To calculate photon differentials from $\mathbf{C}$, we can find two vectors $\Delta\mathbf{p_1}$ and $\Delta\mathbf{p_2}$ satisfying

$$\mathbf{C} = \begin{pmatrix} \Delta\mathbf{p_1} & \Delta\mathbf{p_2} \end{pmatrix} \begin{pmatrix} \Delta\mathbf{p_1} & \Delta\mathbf{p_2} \end{pmatrix}^T. \tag{30.12}$$

But, such a parameterization is not unique. We just assume that $\Delta\mathbf{p_1}$ is along the *x*-axis of the local coordinate frame and solve for $\Delta\mathbf{p_2}$ accordingly.

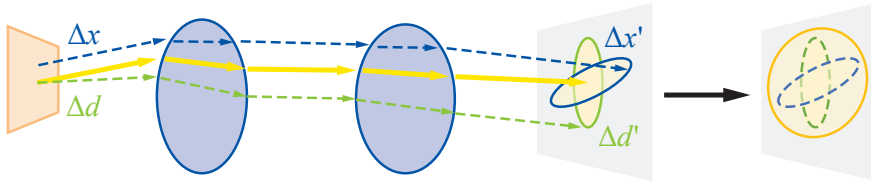
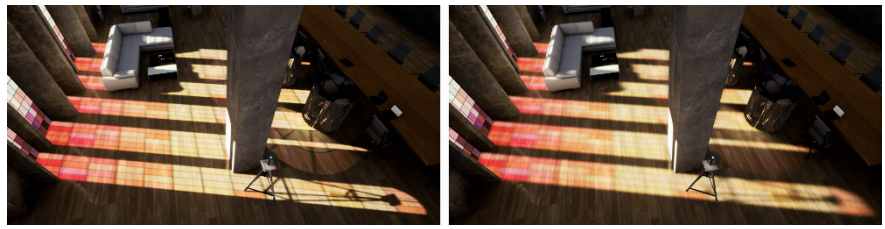


Figure 30-11. *Our soft caustic algorithm tracks photon differentials for both ray position and direction, combines them using convolution and generates a two-dimensional photon footprint.*



(a) *Soft caustics disabled* **(b)** *Soft caustics enabled*

Figure 30-12. *Use soft caustics to simulate soft transparent shadows. (a) The colorful shadow is produced by caustics with softness set to 0. (b) With softness set to 0.15, the shadow is softened. Notice that the caustics generated by a metallic cylinder on the right side are also softened.*

SUMMARY First, our soft caustics implementation estimates photon differentials for both positional and directional perturbations. Then, it constructs the covariance matrices $\mathbf{C_p}$ and $\mathbf{C_d}$ from the differentials and adds both matrices to get the covariance matrix $\mathbf{C}$ for the composite footprint. Finally, it calculates the combined photon differentials $\Delta\mathbf{p_1}$ and $\Delta\mathbf{p_2}$ from $\mathbf{C}$ and applies them to the photon. Figure 30-11 shows the process, and Figure 30-12 shows a scene with and without soft caustics.

30.2.6 RESULTS

The AAPS technique can easily achieve high frame rates for real-time usages while producing accurate, noise-free images. By efficiently applying adaptive photon distribution and anisotropic footprints, the total photons emitted can be kept at a much lower level than traditional offline renderers with similar image quality. Figures 30-13 and 30-14 show two scenes running in real time in which the number of photons is around 50,000 to 100,000.

We picked two scenes for a performance test. The first scene is the classic POV-ray glasses scene shown in Figure 30-1, in which all transparent objects

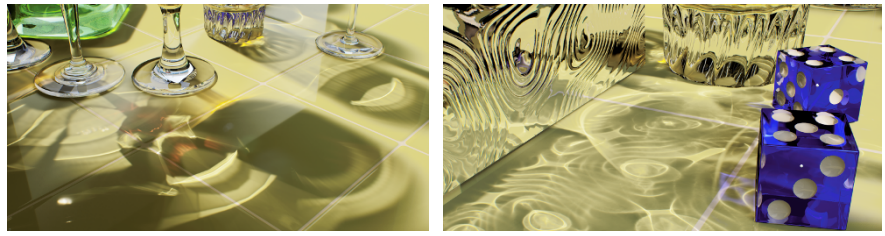


Figure 30-13. *Two views of the POV-ray glasses scene. Left: accurate refractive caustic patterns. Right: reflective caustic patterns cast by normal mapping on the planar metal surface.*

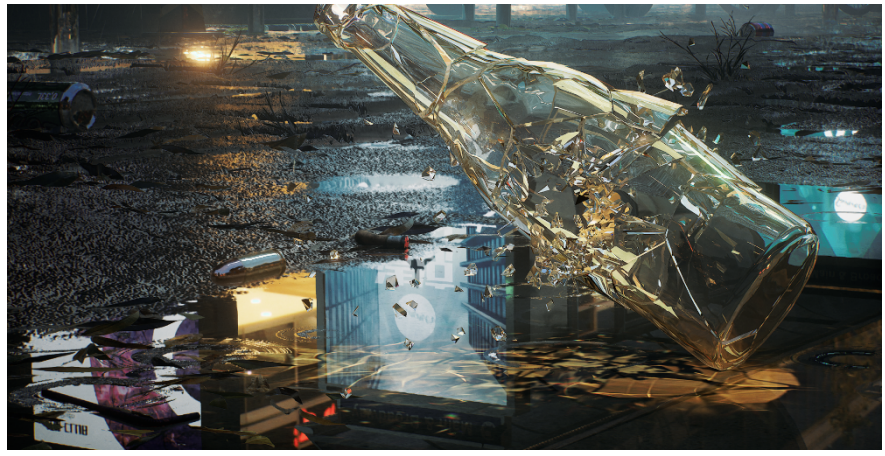


Figure 30-14. *AAPS in real-world applications: a cutscene from the game Bright Memory: Infinite featuring the mesh caustics cast by a shattered glass bottle.*

have both reflected and refracted caustics enabled, and the number of ray bounces is up to 12. The second scene is from the game *Bright Memory: Infinite* and contains a shattered glass bottle (Figure 30-14), where the caustic photons bounce up to eight times before hitting the ground. Beside mesh caustics, both scenes contain large amounts of ray traced refractions and reflections. The tests were performed against 1920×1080 and 2560×1440 resolutions on selected GPUs. All caustics are rendered at full resolutions without any upscaling technique involved. The frame time breakdowns are listed in Figure 30-15.

Based on the performance chart, we can expect that in a usual setup where caustic effects cover a large portion of the view, the cost of rendering caustics



Figure 30-15. *Rendering time breakdown for the POV-ray glasses (Figure 30-1) and Bright Memory (Figure 30-14) scenes at resolutions* 1920 × 1080 *and* 2560 × 1440.

is on the level of 3–4 ms on a performance GPU (RTX 2060) and 1–2 ms on an enthusiast GPU (RTX 3090). These data also show that the cost of caustics is fractional in comparison to other ray tracing regimes. For example, the multi-bounce reflection and refraction on the glass objects take more than 60% of the frame time but are essential for visual quality in a caustic-rich scene. That said, for better performance, the user may have to put more effort toward finding the acceptable appearance of translucent materials other than tweaking caustic parameters.

Further investigation based on profiling tools shows that the AAPS process has two main computational hot spots:

- *Photon tracing:* In the POV-ray glasses scene, the photon tracing stage takes roughly 60% of the caustic rendering time, in which photons bounce up to 12 times before reaching their final hit points. To keep this part of cost under control, the key is to reduce the unnecessary photons, such as excluding materials that can only cast obscure caustics, finding out the maximum acceptable target photon footprint area, culling out photons whose energy falls under a certain threshold, and so on.

- *Photon scattering:* The scattering takes about 20% of the caustic rendering time, which blends all photon footprints into the caustics buffer. To reduce the overhead of blending, the resolution of the caustics buffer can be downscaled to quarter screen size; or consider using an upscaling technique, for example, Deep Learning Super Sampling (DLSS) to boost the frame rate.

30.2.7 LIMITATIONS

In our implementation, the photon tracing step does not use the roughness value in a physically accurate way, thus caustics are still sharp for rough surfaces. We assume that a surface has zero roughness value when calculating photon differentials. This drawback can be relieved by modifying the differentials according to the roughness term. The main challenge is to construct a good approximation to capture reflection and refraction lobes well.

Reflected and refracted caustics are generated in separated threads, which means that having both types of caustics for one object will nearly double the cost of photon tracing.

The calculation of dispersion is not based on continuous spectrum data, thus it is not physically accurate.

Caustic effects seen through reflection and refraction are limited to screen space. For example, if a mirror is placed near a surface that receives caustics, we can only observe the caustics that fall in the current main viewport through the mirror.

The support for area light sources is not optically accurate. For performance consideration, the "soft caustics" cast by area light sources are done by modifying the photon differentials on both ray origins and directions, which means it may not produce the correct contact hardening look in the way of ray traced soft shadows.

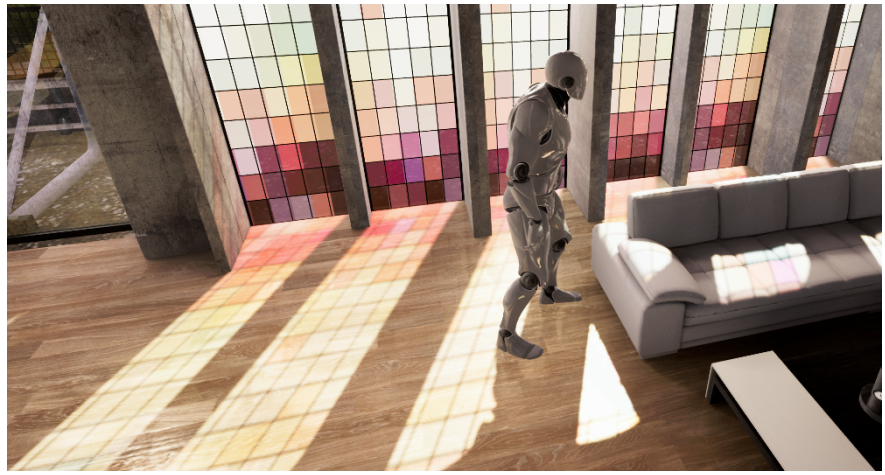


Figure 30-16. *Extended usage of AAPS: simulating transparent shadows. The colorful shadows through the stained-glass windows are one-bounce refractive caustics.*

30.2.8 EXTENDED USAGES

For extended usages other than regular caustic effects cast by glass or metal surfaces, the mesh caustics integration also works for rendering transparent shadows and light spots through textured windows. Figure 30-16 shows a scene with stained glass. The light cast through the windows can be efficiently simulated as a one-bounce refractive caustic. The cost is minimal even if the effect covers a large portion of the screen.

30.3 RAY-GUIDED WATER CAUSTICS

Water caustics have the characteristics of being highly dynamic and interactive, usually covering large areas, and only requiring one-bounce light reflection or refraction, all of which lead us to research in specialized rendering methods. Before describing our latest algorithm improvements, we will give a brief recap to Gruen's method [4]. The method involves mainly two sets of buffers: the *caustics map* stores rasterized water geometry information (positions and normals) from the view of the light source, which may consist of two textures in practice, and the *caustics buffer* accumulates photon footprints in screen space. The workflow consists of four steps:

1. Render water surfaces into the caustics map from the light view, recording the positions and normals of the water surface (Figure 30-17).

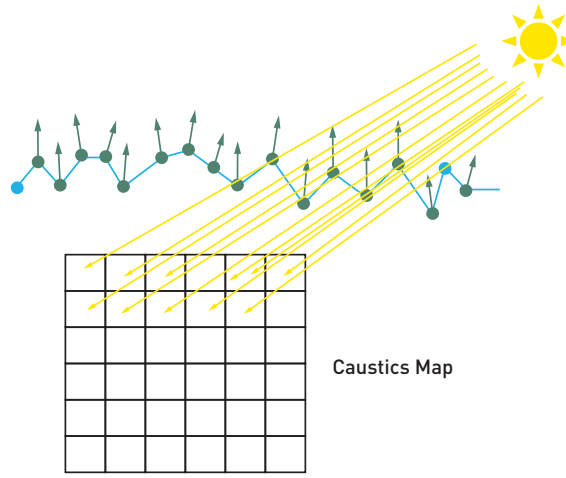


Figure 30-17. *Rendering water geometries into the caustics map from the light view. (From [4].)*

2. Generate rays from positions recorded in the caustics map, and trace them along the reflected or refracted directions calculated from the surface normals. Once the rays hit the scene, record the information of the hit points.

3. Render caustics into the caustics buffer, which is placed in screen space, using the data of the hit points obtained in step 2.

4. Perform denoising on the caustics buffer, and composite the result with the scene.

Our improvements focus on the surface caustics in step 2 and step 3. In step 2, instead of only outputting the intensity of the ray hit point, we count the number of valid hit points and record more data, including position and direction. In step 3, we developed two independent approaches for generating better caustic patterns: *Photon Difference Scattering* (notice that this is not photon differentials), which treats each ray hit point as a photon and renders it as a decal against the scene depth, and *Procedural Caustic Mesh*, which reconstructs the caustic network as a triangular mesh, and then blends it with the scene. As both approaches have pros and cons, users can switch between the two options based on their preference of better performance or higher quality. Besides these overhauls, we also introduce *cascaded caustic maps*, an analog to cascaded shadow maps, to cover mass water bodies by multi-scale rendering.

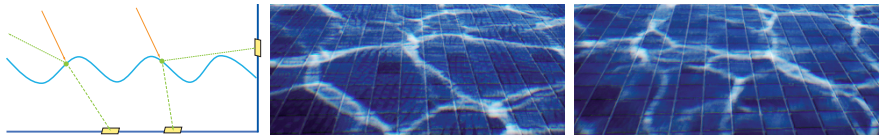


Figure 30-18. *Photon Difference Scattering. Left: creation of quadrilateral photon footprints at ray hit points. Middle: photon footprints visualized as being rendered with equal size; notice the mosaic artifacts. Right: water caustics rendered using PDS with no denoising applied.*

30.3.1 PHOTON DIFFERENCE SCATTERING

The Photon Difference Scattering (PDS) technique, similar to photon differentials, uses the finite difference of nearby rays' data to compute photon coverage, which can give more accurate results than using local perturbations as in photon differentials. Unlike photon differentials, PDS does not need to access the geometry data of the caustics receivers, so it is easier to implement in game engines. With PDS, we can achieve the same quality of the original caustics mapping method by casting much sparser rays, thus greatly improving the ray tracing performance. The brightness adjustment by footprint coverage ensures the correct intensity distribution from all incident angles, while some slope angles may raise artifacts in Gruen's method.

With the PDS approach, we treat ray hit points as photon footprints and render them as decal sprites against the scene depth. Figure 30-18 (middle) shows the photon footprints being rendered at a fixed size, which forms correct caustic envelopes but leaves gaps in between. To fill the scene surface with compact quads, we need to find a proper size for each footprint.

Fortunately, for water caustics each ray is cast from one single reflection or refraction, which means that we can easily backtrace to the ray's origin in the caustics map and access its adjacent rays' origins and directions. The right size of the footprint is then estimated using these ray data around the hit point (Figure 30-19).

The initial size of the footprint is determined by the resolution of the caustics map and the desired precision set by the user, then its scaling factor is derived from the current hit point and the estimated hit points:

$$\text{Scale} = \frac{\text{TriangleArea}(\text{hit0}, \text{hit1}, \text{hit2})}{\text{TriangleArea}(\text{pos0}, \text{pos1}, \text{pos2})}, \tag{30.13}$$

where *hit*0 is the position of the current hit point, *hit*1 and *hit*2 are the

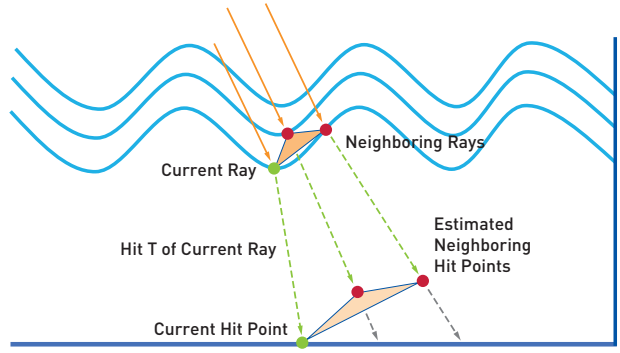


Figure 30-19. *Calculating the size of a photon footprint by combining the ray's origin, direction, and hit point data with the adjacent ray's origins and directions. The hit points of adjacent rays (red) are estimated from the current ray's hit T (green).*

estimated positions of neighboring hit points, and *pos*0, *pos*1, and *pos*2 are the corresponding original positions in the caustics map.

Before finally rendering the photon footprint onto the caustics buffer, the intensity, size, and orientation of the quad sprite are also adjusted by the photon direction and the surface normal: the quad is placed perpendicular to the ray direction and then projected to the opaque surface (Figure 30-20). Because the quad is smeared over the receiving surface, its intensity is cosine weighted by the incident angle. The lighting result is calculated using the scene materials during the scattering.

Choosing a proper shape for footprints can significantly improve the quality. Figure 30-21 shows that the elliptic footprints provide better result than the rectangular ones. In Figure 30-18 (right), after PDS was applied, the footprints formed into continuous patterns without any denoising involved.

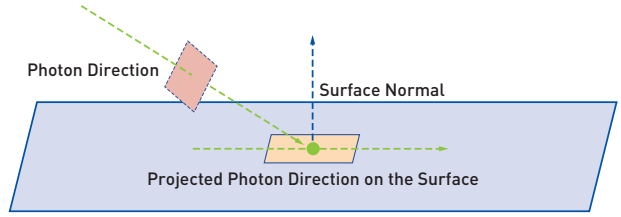
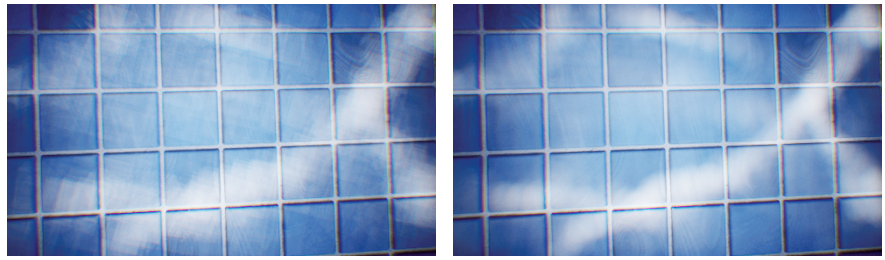


Figure 30-20. *The intensity, size, and orientation of the quad sprite are also adjusted by the photon direction and the surface normal.*



(a) *Rectangular footprints* **(b)** *Elliptic footprints*

Figure 30-21. *Comparison of caustics footprints: (b) elliptic footprints provide sharper and clearer caustics than (a) rectangular footprints.*

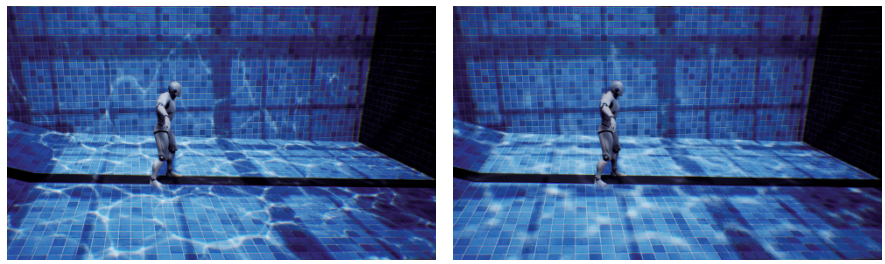


Figure 30-22. *The results of using different caustics map resolutions with photon difference scattering. Left: resolution* 2048 × 2048 *covering a* 30 *m* × 30 *m area. Right: resolution* 512 × 512 *covering the same area.*

On the pro side, PDS can render high-quality water caustics with low cost and can easily extend the supports for many types of light sources, including area light. On the con side, it is sensitive to the caustics map resolution related to the covering range—applying a low-resolution caustics map to a large scene area may result in very blurry caustic patterns (Figure 30-22).

30.3.2 PROCEDURAL CAUSTIC MESH

The other approach for reconstructing caustic patterns is Procedural Caustic Mesh (PCM), which converts hit points into an intermediate mesh: each hit point is mapped to a vertex in the mesh whose topology is a triangle list that maps to the regular grids in the caustics map. After the ray tracing pass, a compute shader fetches the hit point data, evaluates the contribution and intensity of each primitive according to its world-space area, discards invalid primitives, and generates the index buffer. The mesh is then rendered onto the caustics buffer in a rasterization pass (Figure 30-23). In practice, we build

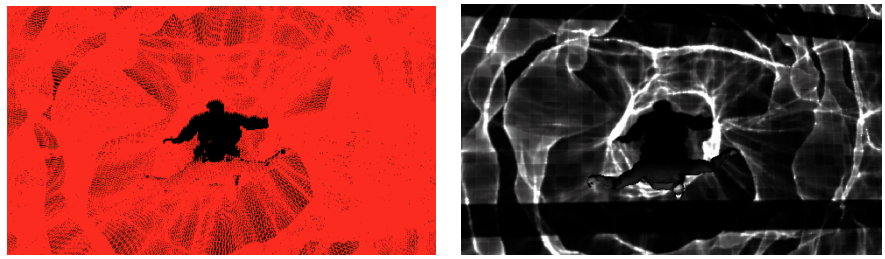


Figure 30-23. *Procedural Caustic Mesh. Left: the refracted caustic mesh; notice that the shadow area is culled from the mesh. Right: the rendered mesh modulated by scene materials.*

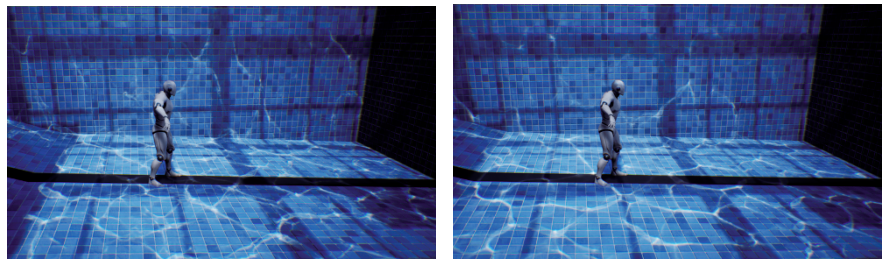


Figure 30-24. *The results of using different caustics map resolutions with PCM. Left: resolution* 2048×2048 *covering a* $30\ m \times 30\ m$ *area; Right: resolution* 512×512 *covering the same area. Notice that the qualities of the two are almost the same.*

two procedural caustic meshes for every water object, one for reflection and the other for refraction.

The advantage of this approach is that it always produces sharp caustic patterns even if the caustics map resolution is very low (Figure 30-24). However, it also raises the "black edge" artifact at object corners where the mesh triangles span over the culling region (Figure 30-25). For the best result, users can choose between PDS and PCM for better quality. In general, PDS is more flexible and well suited for water areas in a confined space like swimming pools, whereas PCM is more efficient when coupling with large water bodies, such as lakes and oceans.

30.3.3 CASCADED CAUSTICS MAPS

To generate caustics for large water bodies like ocean surfaces, we have implemented *cascaded caustics maps* (CCM) that work in the same way as cascaded shadow maps (CSM). Figure 30-26 shows a configuration of four

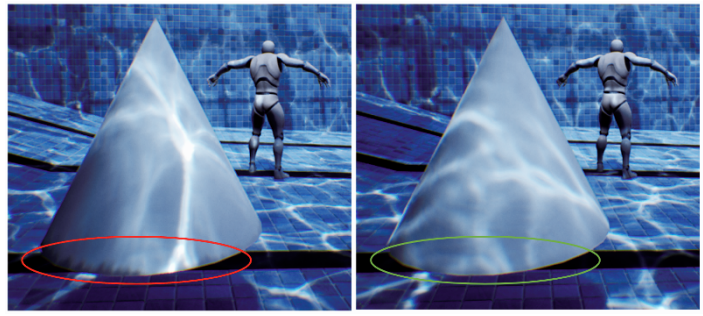


Figure 30-25. *Left: artifact raised by PCM along the discontinuous edge of the scene geometry due to inaccurate culling of mesh primitives. Right: in comparison, PDS has no such artifact.*



Figure 30-26. *Left: a CCM configuration of four cascades. Right: an aerial view of the ocean surface with reflective and refractive caustics rendered by CCM.*

cascades, where each cascade contains a caustics map at a user-selected resolution. CCM can be coupled with PDS to mitigate blurry results when rendering with limited photon budget but higher details are desired at near sight, as it allows us to distribute more photons at the innermost cascade to lift the quality and less photons at the outer cascades to keep the cost down.

Unlike CSM, to implement cascaded maps in a ray tracing pipeline, we need to determine the number of shader threads for dispatching rays and the scheme of assigning data to the threads. The following formula gives the threads needed for the CCM:

$$N_{\text{thread}} = W \times H \times \left[1 + \left(1 - \frac{1}{\text{Scale}^2}\right) \times (N_{\text{level}} - 1)\right], \tag{30.14}$$

where W and H are the width and height of the caustics map, Scale is the

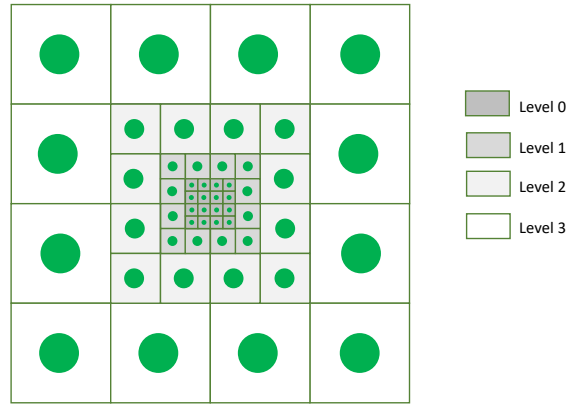


Figure 30-27. *Assigning CCM data to ray generation shader threads: A setup with four cascades, each cascade at* 4×4 *resolution, with size scaling* 2 *between the cascades, where the gray areas are overlapped. The cascaded setup only dispatches* 52 *threads in total, while a uniform setup requires* 1024 *threads.*

length ratio between two adjacent cascade levels, and N_{level} is the number of cascades. Figure 30-27 demonstrates an example on the mapping between the CCM and the shader threads.

30.3.4 SOFT WATER CAUSTICS BY AREA LIGHTS

We have extended the PDS method to simulate soft water caustics cast by area light sources. Currently, only textured rectangular lights are supported, but the technique can be easily expanded to accommodate more complicated area lights. Applying a 2D texture to a rectangular light to define the surface intensity and to emit photons accordingly allows us to simulate any planar luminaries. Unlike with other light types, the photons leaving an area light are going in all directions (Figure 30-28). Thus, water caustics exhibit "softness" in the way of soft shadows.

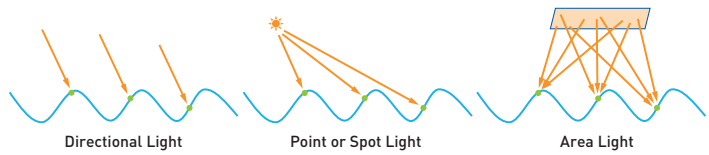


Figure 30-28. *Only one ray hits each point on the water surface with directional, point, and spot lights. But with an area light, each point on the water surface may be hit by many rays.*

To simulate this soft effect in real time, we developed a temporal method to reuse the caustic patterns over multiple frames. It consists of two steps:

1. For each valid point in the caustics map in the current frame, we select an incident ray from a random sampling point on the rectangular light, and we trace it through the water surface.
2. We accumulate the caustics over several frames with a temporal filter to output the soft caustics.

Performance-wise, we can produce acceptable soft water caustics by any rectangular lights at merely the same cost as other light types.

30.3.5 RESULTS

The RGWC technique rasterizes the water mesh into a caustics map, avoiding ray tracing between light sources and water surfaces, which greatly helps the rendering efficiency. Meanwhile, the two caustic reconstruction methods, PDS and PCM, allow the creation of high-quality patterns while covering large water bodies. In addition, CCM and soft water caustics greatly expand the usability of RGWC.

The performance of RGWC is affected by many factors, such as the resolution of the caustics map and the caustics buffer, the water coverage in the view, and the number of light sources that affect the caustics. We performed some testing using the two scenes shown in Figure 30-29.



Figure 30-29. *Two scenes selected for RGWC performance tests. Left: swimming pool lit by one directional light source. Right: seaside town lit by one directional light and having a four-cascade CCM configuration. Both scenes have above and under water caustics enabled. The resolution of the caustics buffer is set to full screen size. The caustics map is set to* 1024×1024.

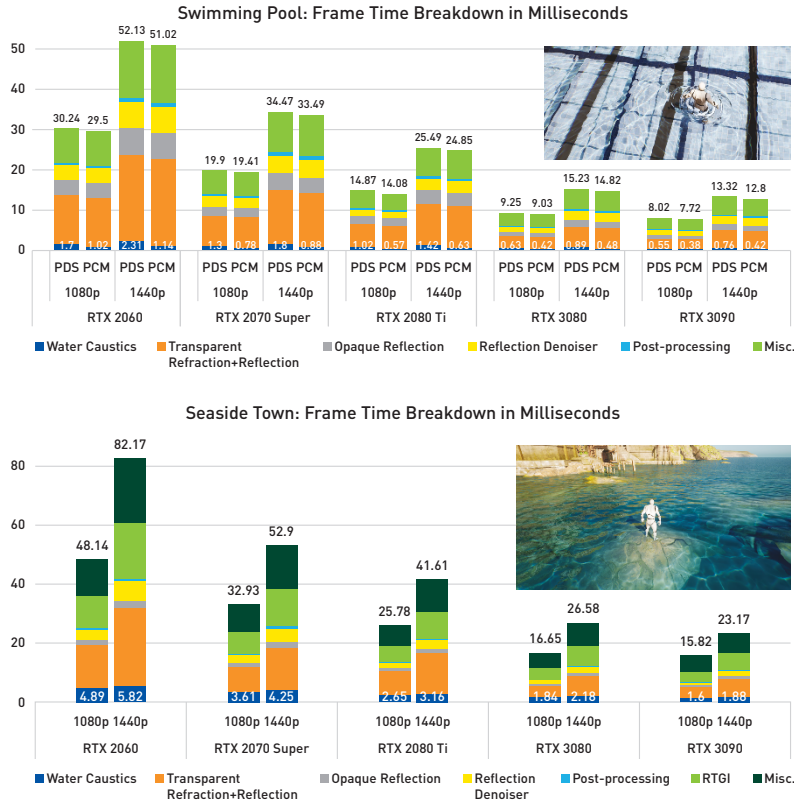


Figure 30-30. *Water caustics rendering time breakdown for the scenes Swimming Pool (top) and Seaside Town (bottom) at screen resolutions* 1920×1080 *and* 2560×1440.

Figure 30-30 (top) compares the performance of PDS and PCM on selected GPUs using the test scene Swimming Pool (Figure 30-29, left). In general, PCM is faster than PDS. Further investigations using profiling tools reveal that part of the overhead of PCM comes from the workload that reconstructs the new index buffer, which is dependent on the vertex number (same as the size of the caustics map) but less dependent to the size of the caustics buffer. And rendering the mesh avoids rendering overlapping sprites in PDS. This explains why PCM is more efficient than PDS and less impacted by the screen resolutions.

Figure 30-30 (bottom) lists the cost of CCM using the test scene Seaside Town (Figure 30-29, right). The scene has four cascades of a $1024 \times 1024 \times 4$ caustics map to cover the sea surface of 240×240 m^2. Because CCM does not work with PCM, only PDS numbers are shown here. The usage of CCM

significantly increases the cost of RGWC because all four caustics maps are ray traced. However, the aerial view of the sea caustics cannot be easily handled by PCM. Thus, CCM is the only option here to produce high-quality caustic patterns for large scene coverage.

30.3.6 LIMITATIONS

In our implementation, we only render the reflected and refracted caustics generated by the first bounce of photons. Photons bouncing multiple times among the water ripples are not captured.

CCM only works for directional lights and does not support PCM.

Water caustics cannot be seen within ray traced reflections. For example, when underwater, the total internal reflection on the water surface does not show the underwater caustics.

30.4 CONCLUSION

In this chapter, we introduced two techniques to render caustics effect for translucent or metallic objects and water surface. Adaptive Anisotropic Photon Scattering can produce high-quality caustics effects with a reasonable number of rays each frame, which insures the high performance required by many games. As a bonus effect, this technique can also be used to produce shadows cast by translucent objects, such as colored glass windows. Ray Guided Water Caustics is a highly specialized technique to simulate one-bounce water caustics, being very efficient and versatile to handle all sorts of water bodies. Also, it does not need to track light propagation with photon differentials that require additional hit shader code in all materials, which makes it easy to integrate into commercial products.

We have integrated both techniques into NVIDIA's customized Unreal Engine 4 branch. The source code can be obtained at the repository [9].

ACKNOWLEDGMENTS

We thank Nan Lin for providing reference materials and helping edit this chapter. Thanks to Jiaqi Wang for his numerous work on creating and polishing test scenes, performance tuning, and rendering quality control. Thanks to Tao Shi for measuring performance statistics on various hardware and configurations. We would also thank Evan Hart for his advice on CPU-side performance optimization.

REFERENCES

[1] Boksansky, J., Wimmer, M., and Bittner, J. Ray traced shadows: Maintaining real-time frame rates. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 159–182. Apress, 2019.

[2] Elek, O., Bauszat, P., Ritschel, T., Magnor, M., and Seidel, H.-P. Spectral ray differentials. 33(4):113–122, 2014. DOI: 10.1111/cgf.12418.

[3] Estevez, A. C. and Kulla, C. Practical caustics rendering with adaptive photon guiding. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Talks*, pages 1–2, 2020.

[4] Gruen, H. Ray-guided volumetric water caustics in single scattering media with DXR. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 183–201. Apress, 2019.

[5] Igehy, H. Tracing ray differentials. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 179–186, 1999. DOI: 10.1145/311535.311555.

[6] Jensen, H. W. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, 2001.

[7] Kim, H. Caustics using screen-space photon mapping. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 543–555. Apress, 2019.

[8] Mara, M., Luebke, D., and McGuire, M. Toward practical real-time photon mapping: Efficient GPU density estimation. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 71–78, 2013. DOI: 10.1145/2448196.2448207.

[9] NVIDIA. NVIDIA RTX experimental branch of Unreal Engine 4. https://github.com/NVRTX/UnrealEngine/tree/NvRTX_Caustics-4.26, 2020. (Registration required to access link. See https://developer.nvidia.com/unrealengine).

[10] Schjoth, L., Frisvad, J. R., Erleben, K., and Sporring, J. Photon differentials. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, pages 179–186, 2007. DOI: 10.1145/1321261.1321293.

[11] Wyman, C. and Nichols, G. Adaptive caustic maps using deferred shading. *Computer Graphics Forum*, 28(2):309–318, 2009. DOI: 10.1111/j.1467-8659.2009.01370.x.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 31

TILT-SHIFT RENDERING USING A THIN LENS MODEL

Andrew Kensler
Amazon, formerly Pixar Animation Studios

ABSTRACT

A tilt-shift lens is a lens whose longitudinal axis can be moved out of alignment with the center of the film or image sensor—either by rotation for tilt or translation for shift. In this chapter, we will review the classic thin lens camera model in ray tracing, which sits at the sweet spot between the simple pinhole model and the complex lens systems. Then we will show how to extend it to model a tilt-shift lens. We will also show how to solve for the placement of the lens to achieve an arbitrary plane of focus.

31.1 INTRODUCTION

Tilted lenses frequently have shallow apparent depths of field, which make them popular for creating a miniaturized look (e.g., Figure 31-1a). Though it is common to fake this look with Gaussian blurs masked along a gradient, the real effect can be subtle and much more interesting. See https://www.reddit.com/r/tiltshift/ for many examples, both real and faked.



(a) *Tilted lens* **(b)** *Tilted and shifted lens*

Figure 31-1. *An example scene. (a) Miniaturized appearance with tilted lens. (b) Combining lens tilt and shift. Lens shift is used so that the center of perspective is in the upper left, and lens tilt has been applied so that the plane of focus is perpendicular to the camera and aligned to emphasize the building fronts on the right. Unlike with a standard lens, all of the fronts are in sharp focus regardless of distance down the street.*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_31

On the other hand, lenses that can be shifted off-center can be used to correct for perspective distortion (also known as the keystoning effect), and to switch between one-, two-, and three-point perspective.

Commonly, these abilities are combined in a tilt-shift lens (e.g., Figure 31-1b). Early examples of these could be found in the view camera where the lens and the film are held in place at opposite ends of a flexible bellows. For details, see Merklinger [2]. For an overview of photography, refer back to Chapter 1.

In this chapter, Section 31.2 will go over the traditional thin lens model in rendering, first introduced to computer graphics by Potmesil and Chakravarty [3] and to ray tracing by Cook et al. [1]. Then, Section 31.3 will cover how to add lens shift to it, while Section 31.4 will address how to modify it for tilted lenses. Since direct control is surprisingly finicky, Section 31.5 will address how to solve for the correct lens and sensor positions to achieve an arbitrary plane of focus. Finally, Section 31.6 will show some more examples of tilted lens rendering and some interesting consequences.

31.2 THIN LENS MODEL

For this chapter, we will assume that everything is in camera space, with $+\hat{\mathbf{z}}$ pointing down the view direction and the lens centered on the origin. The sample code is written for a left-handed Y-up camera coordinate system. Some distance ahead of the lens is an object at a point P that we would like to image to a point P' on the sensor behind the lens. There is an aperture that restricts the size of the cone of light passing through the lens. See Figure 31-2.

The focal length of the lens, f, and the perpendicular distances between points on the lens, object, and image planes are all related by Gauss's thin lens equation, where p is the distance from the lens to the object and p' is the distance from the lens to the image plane:

$$\frac{1}{f} = \frac{1}{p} + \frac{1}{p'}. \tag{31.1}$$

We can also reformulate the thin lens equation to compute one distance if we know the other. In this case, we can determine how far back from the lens a given point will focus to. The sensor must be placed here in order for the point to be in focus:

$$p' = \frac{pf}{p - f}. \tag{31.2}$$

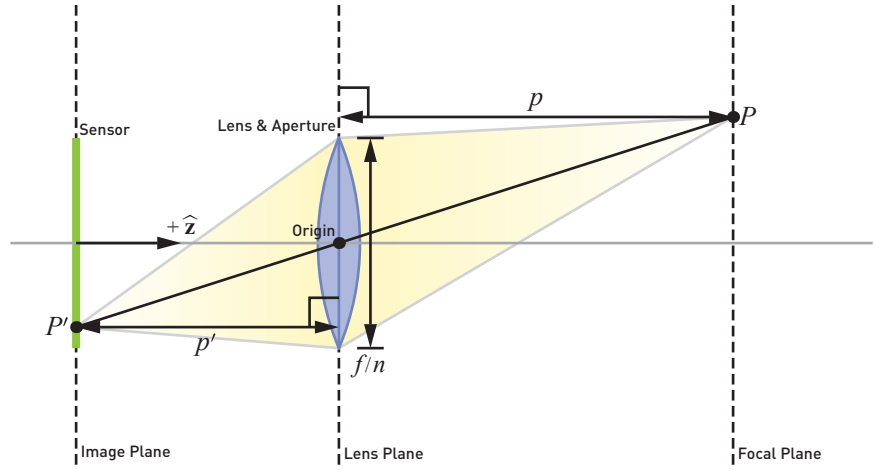


Figure 31-2. *The geometry of the thin lens model.*

Note that we must be very careful with signs here: for a lens projecting the object to a real image such as this, the focal length and two distances are positive, while the z-coordinates of points are negative when they are behind the lens.

If an object point is off of the focal plane, then it will focus to a point off of the image plane. With a circular aperture, this creates an image of the aperture on the sensor called the *circle of confusion*. The size of the acceptable circle of confusion determines the near and far limits around the focal plane for what is considered in-focus. The distance between these two limits defines the depth of field. See Figure 31-3. These near and far limits also form planes. Note that with a standard lens setup, the image plane, lens plane, focal plane, and near and far focal planes are all parallel.

One further assumption of the thin lens model is that light between an object and its image passes through the center of a thin lens undeflected. Consequently, the coordinates of the two points in camera space are related by a simple scaling factor.

To render with this, we first choose a desired focal distance for where we want the focal plane to be and use it as p in Equation 31.2. Define the distance between the lens center and the image plane with the sensor as the result of that equation, $s = p'$. Then for each pixel in an image, we simply take the location of the pixel on our sensor, P' (where $P'_z = -s$), and map it to the point

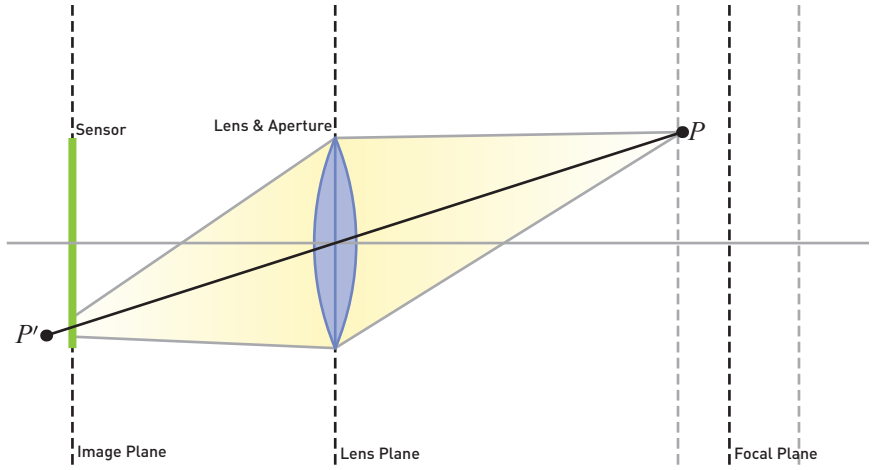


Figure 31-3. *Defocus and the circle of confusion. The object projects to a point behind the sensor and so a small circle of light from it impinges on the sensor.*

on the focal plane, P, that focuses there (note that the scaling factor relating them is constant):

$$P = P' \frac{f}{f - s}. \tag{31.3}$$

Then we aim a ray toward that point, in direction $\widehat{\mathbf{d}}$ from a randomly sampled point O within the lens aperture (Figure 31-4).

The following GLSL ES (WebGL) sample code puts this all together. It takes as input a screen coordinate and a pair of random numbers. The screen coordinate is the pixel position normalized so that the short edge lies in the range [–1, 1]. The sensor size is then the size of this short edge. This makes the field of view angle implicit on the sensor size, the focal distance, and the focal length. Though we could specify a desired field of view and then derive the sensor size to use instead, physical cameras have constant sensor sizes and so adjusting the focal distance changes the field of view angle. This is sometimes called *focus breathing* and is modeled here.

The random numbers given should be uniformly distributed within the $[0, 1]^2$ square. Ideally, these would be generated using quasi-Monte Carlo for quicker convergence. Note that in the sample code here we use them to uniformly sample a circular disk for the lens position, which yields perfectly flat, circular bokeh blurs. Alternately, we could use them to sample a polygon, sample with an uneven distribution, or even sample a density image with

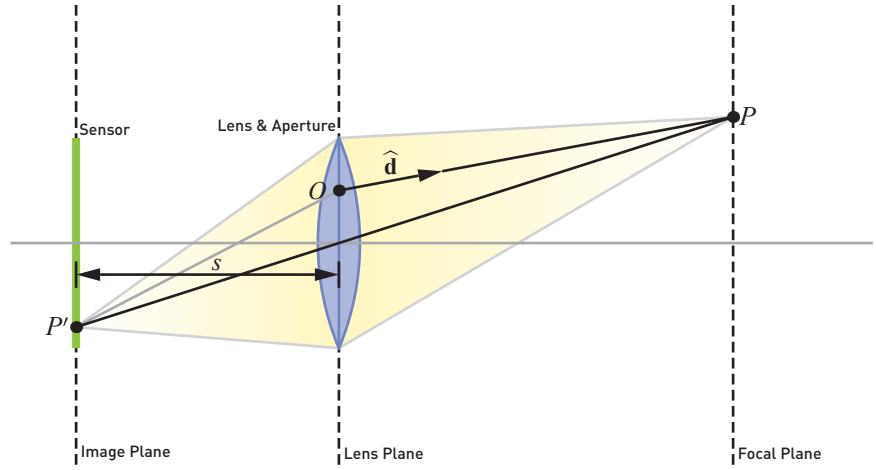


Figure 31-4. *Sampling the lens with a ray. The ray is traced from a random position on the lens to where the pixel on the sensor projects onto the focal plane.*

rounded iris blades, dust, and scratches. The choice of distributions will directly determine the appearance of the bokeh.

```
void thin_lens(vec2 screen, vec2 random,
               out vec3 ray_origin, out vec3 ray_direction)
{
    // f  : focal_length      p : focal_distance
    // n  : f_stop            P : focused
    // s  : image_plane       O : ray_origin
    // P' : sensor            d : ray_direction

    // Lens values (precomputable)
    float aperture = focal_length / f_stop;
    // Image plane values (precomputable)
    float image_plane = focal_distance * focal_length /
        (focal_distance - focal_length);

    // Image plane values (render-time)
    vec3 sensor = vec3(screen * 0.5 * sensor_size, -image_plane);
    // Lens values (render-time)
    float theta = 6.28318531 * random.x;
    float r = aperture * sqrt(random.y);
    vec3 lens = vec3(cos(theta) * r, sin(theta) * r, 0.0);
    // Focal plane values (render-time)
    vec3 focused = sensor * focal_length /
        (focal_length - image_plane);

    ray_origin = lens;
    ray_direction = normalize(focused - lens);
}
```

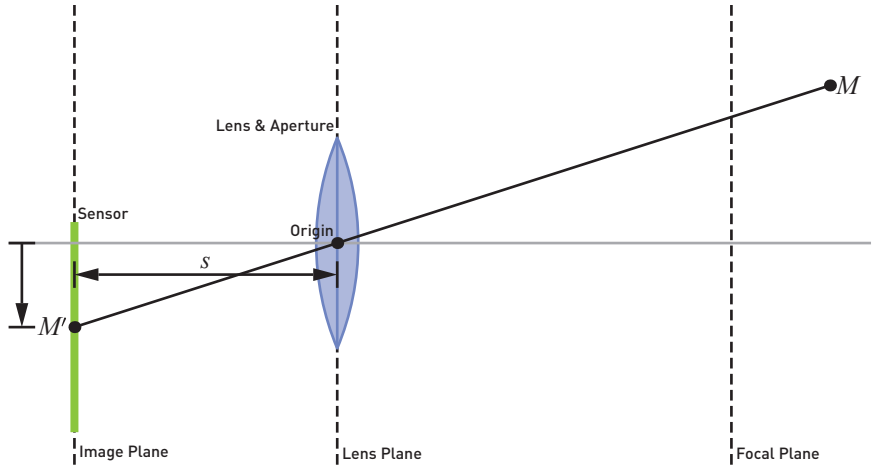


Figure 31-5. *Translating the sensor for lens shift. The point M that is shifted to appear in the image center does not need to be on the focal plane.*

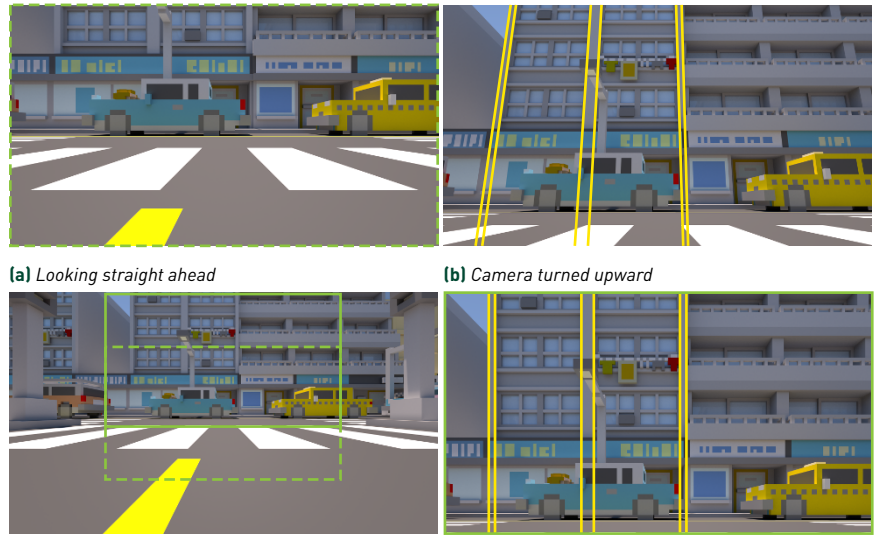
31.3 LENS SHIFT

Now consider adding lens shift to this model. Since this means that the center of the sensor is no longer in alignment with the center of the lens, we could model this as a translation of the lens within the lens plane. However, it is easier to keep the lens centered at the origin in camera space and instead translate the sensor within the image plane (Figure 31-5). This corresponds to a simple addition to the x- and y-coordinates (but not the z-coordinate) of P' for a pixel on the sensor before we map it to P on the focal plane and project a ray through it as before.

If there is a point M that we would like to shift to be centered in the middle of the image, then we can project it through the center of the lens and onto the image plane at M'. Then $(M'_x, M'_y, 0)$ gives the translation needed:

$$M' = M\frac{-s}{M_z}. \tag{31.4}$$

Commonly, this effect is used for architectural photography. As an example, suppose that we have a picture near street level and we would like to look at things a little higher up. When looking straight ahead level (Figure 31-6a), the upright lines on the vertical elements are parallel. However, if we just swivel or turn the camera upward, this creates a foreshortening and causes the lines on the upright elements to converge rather than remain parallel as before



(a) *Looking straight ahead*

(b) *Camera turned upward*

(c) *Larger image with wider angle*

(d) *Lens shift*

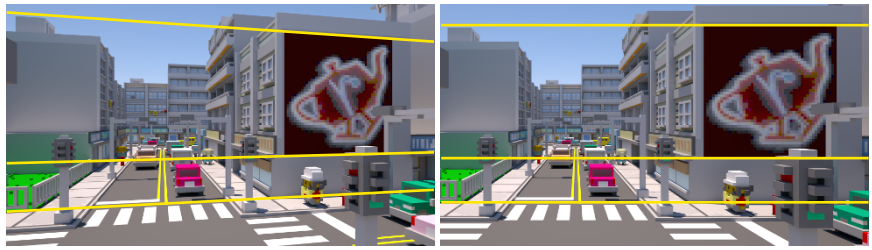
Figure 31-6. *Looking upward. (a) The original image, looking straight ahead. (b) Vertical lines are no longer parallel if the camera is turned to look upward. (c) Shifting as an off-center crop of a larger image; compare the solid and dashed green rectangles. (d) Shifting allows a higher view while keeping vertical lines parallel.*

(Figure 31-6b). This is a form of perspective distortion and is frequently called the *keystone effect*.

Instead, we can keep the view direction level as before and shift the sensor down relative to the lens. Alternatively, we can think of this as rendering to a larger image with an equivalently wider field of view (or zoomed out) and then cropping back to the original image size but off-center (Figure 31-6c).

This still allows us to see things higher up as with turning the camera, but now the lines on upright elements in our scene remain vertical (Figure 31-6d). With this, the keystoning or perspective distortion has been corrected.

Of course, we can also do the same thing horizontally. Here we have an example with the camera over the center of the street but turned to the right to avoid cutting off the billboard (Figure 31-7a). Instead, we can keep the camera pointed straight down the street and shift the sensor to the left so that we can capture more of the scene on the right side while keeping the horizontal lines parallel (Figure 31-7b).



(a) *Camera turned toward the right* **(b)** *Shifting toward the right*

Figure 31-7. *Looking to the side. (a) Turning toward the right. (b) Shifting toward the right.*

One other thing that this does is to simplify the two-point perspective of the first view down to a classic single-point perspective in the second. All of the converging lines now converge to a single point, which can be off-center. Careful use of shifting can also reduce a three-point perspective down to two-point while keeping the subject of a scene framed.

In view camera terms, a vertical shift is a "rise" or "fall," while a horizontal shift may be a "shift" or "cross."

31.4 LENS TILT

Next consider the case of a tilted lens. In view camera terms, this will be "swing" when horizontal and "tilt" when vertical. Of course, the two may be combined, and here the mathematics gets a little more complicated.

To model this, we will keep the lens centered at the origin in camera space, but the lens plane is now aligned with the unit normal vector $\hat{\mathbf{t}}$. See Figure 31-8.

Recall that the thin lens formula relates distances between the lens plane and the object and image points. These are perpendicular distances, so we need to use the scalar projections of the points onto vector $\hat{\mathbf{t}}$ (with appropriate sign corrections) to compute these distances.

Consequently, the mapping between a pixel position on the sensor, P', and the corresponding point on the focal plane, P, is still done through a scale factor (as in Equation 31.3), but now it is no longer constant:

$$P = P' \frac{f}{f + P' \cdot \hat{\mathbf{t}}}. \tag{31.5}$$

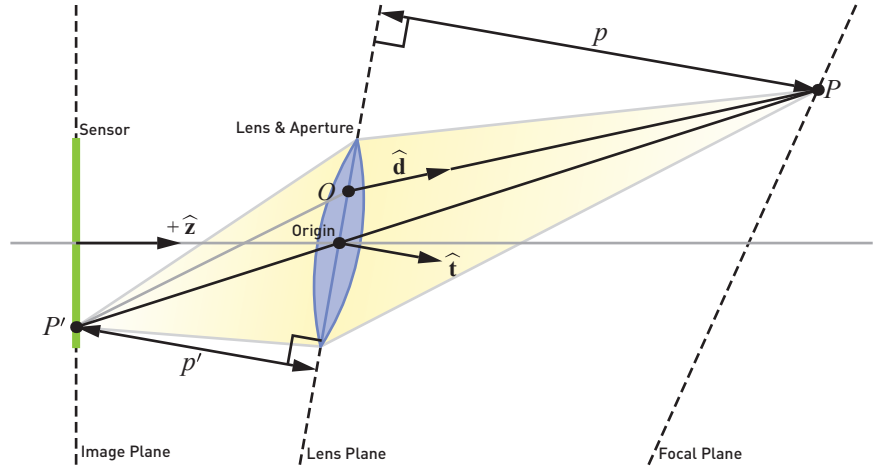


Figure 31-8. *The geometry of a tilted lens.*

Sampling the points on the lens to find the ray origin O is now slightly more complex and requires that we build an orthonormal basis in camera space around the tilt direction $\hat{\mathbf{t}}$.

Finding the ray direction remains mostly as before. However, because the distances between the planes are now varying when the lens is tilted, the focal plane and lens plane can now intersect. As a result, some points on the sensor may map to points that are behind the lens plane. When $\hat{\mathbf{t}} \cdot P < 0$, we have a virtual image. In this case, the rays diverge outward from the point behind the lens, and we must flip $\hat{\mathbf{d}}$ so that the ray goes forward into the scene.

The optics of the tilted lens has some interesting implications. The *Scheimpflug principle* states that when the image, lens, and focal planes are not parallel, then they all form a sheaf of planes that intersect along a common line, typically called the *Scheimpflug line* (Figure 31-9). Furthermore, the depth of field becomes a wedge, emanating from a second line, sometimes called the *hinge line*. As the sensor moves closer to or farther from the lens, the focal plane will pivot around this hinge line, forming a secondary sheaf of planes.

31.5 DIRECTING THE TILT

Because of all this, we have the flexibility to place the plane of focus nearly anywhere. But actually doing this through direct control of the lens tilt can be

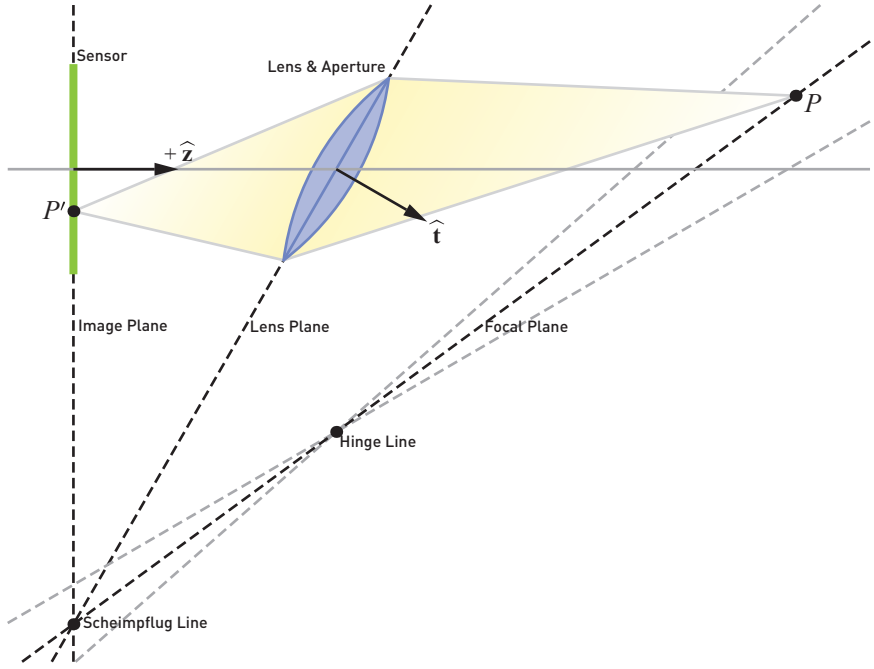


Figure 31-9. *The Scheimpflug principle and hinge line. The image, lens, and focal planes all intersect at a common line, which is shown here perpendicular to the page. The hinge line is also perpendicular to the page.*

tricky. Photographers tend to resort to tedious trial and error. In the digital world we can do better.

Suppose that we have three points, A, B, and C, defining the plane that we would like to have in focus (Figure 31-10). This plane has the unit normal vector $\hat{\mathbf{n}}$. How can we solve for the lens tilt $\hat{\mathbf{t}}$ and the distance s from the lens center to the image plane?

We know that points A and B must focus to points at the same z-coordinate behind the lens, which from the Equation 31.5 gives us s:

$$s = A_z \frac{f}{A \cdot \hat{\mathbf{t}} - f} = B_z \frac{f}{B \cdot \hat{\mathbf{t}} - f}. \tag{31.6}$$

Taking the right-hand sides, cross multiplying, expanding, and collecting gives

$$(A_z B_x - A_x B_z) t_x + (A_z B_y - A_y B_z) t_y = (A_z - B_z) f. \tag{31.7}$$

We also know from the Scheimpflug principle that the image plane, focal

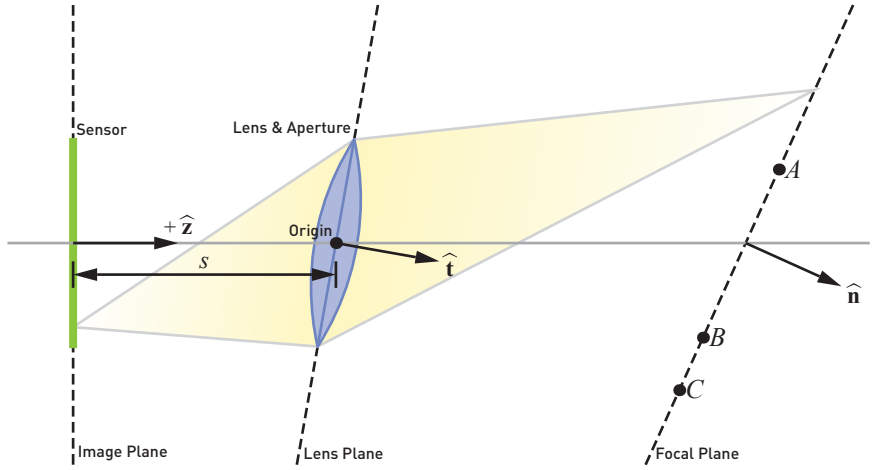


Figure 31-10. *Directing the lens tilt. Points A, B, and C are used to define the desired focal plane and appear collinear in a 2D projection such as this when the Scheimpflug line is drawn perpendicular to the page.*

plane, and lens plane must intersect at a line in common, which we can express as

$$\hat{\mathbf{z}} \cdot (\hat{\mathbf{n}} \times \hat{\mathbf{t}}) = 0. \tag{31.8}$$

Expanding this allows us to compute t_y in terms of t_x:

$$t_y = t_x n_y / n_x. \tag{31.9}$$

Taking Equation 31.9 and substituting it into Equation 31.7 allows us to solve for the x-component of the tilt, and from there we can compute the rest. Because our premises assumed that $\hat{\mathbf{t}}$ was scaled to unit length, the z-component must complete the unit vector. The full solution to the tilt of the lens and the placement of the sensor is thus:

$$t_x = \frac{(A_z - B_z)f}{A_z B_x - B_z A_x + (A_z B_y - B_z A_y) n_y / n_x}, \tag{31.10}$$

$$t_y = t_x n_y / n_x, \tag{31.11}$$

$$t_z = \sqrt{1 - t_x^2 - t_y^2}, \tag{31.12}$$

$$s = A_z \frac{f}{A \cdot \hat{\mathbf{t}} - f}. \tag{31.13}$$

In some cases, depending on the desired plane of focus, $|n_x|$ may become quite small or even be zero. To avoid division by zero or other numeric

precision problems, the x- and y-components can be exchanged by symmetry if n_x is closer to zero than n_y. However, if both n_x and n_y are zero, then the plane of focus is parallel to the image and lens planes and there is no lens tilt. In this case, setting t_x and t_y to zero before computing t_z and s produces the correct results, and the tilted lens model reduces to the standard thin lens model.

Combining both the shift and tilt extensions to the thin lens camera model, together with the solutions to find the tilt and the shift positions, yields the following implementation of the full tilt-shift model:

```
void tilt_shift(vec2 screen, vec2 random,
                out vec3 ray_origin, out vec3 ray_direction)
{
    // n  : normal      A : focus_a
    // t  : tilt        B : focus_b
    // M  : middle      C : focus_c
    // M' : shift

    // Focal plane values (precomputable)
    vec3 normal = normalize(cross(focus_b - focus_a,
                                  focus_c - focus_a));
    // Lens values (precomputable)
    vec3 tilt = vec3(0.0);
    if (abs(normal.x) > abs(normal.y))
    {
        tilt.x = (focus_a.z - focus_b.z) * focal_length /
            (focus_a.z * focus_b.x - focus_b.z * focus_a.x +
             (focus_a.z * focus_b.y - focus_b.z * focus_a.y) *
             normal.y / normal.x);
        tilt.y = tilt.x * normal.y / normal.x;
    }
    else if (abs(normal.y) > 0.0)
    {
        tilt.y = (focus_a.z - focus_b.z) * focal_length /
            (focus_a.z * focus_b.y - focus_b.z * focus_a.y +
             (focus_a.z * focus_b.x - focus_b.z * focus_a.x) *
             normal.x / normal.y);
        tilt.x = tilt.y * normal.x / normal.y;
    }
    tilt.z = sqrt(1.0 - tilt.x * tilt.x - tilt.y * tilt.y);
    vec3 basis_u = normalize(cross(tilt,
        abs(tilt.x) > abs(tilt.y) ? vec3(0.0, 1.0, 0.0)
                                  : vec3(1.0, 0.0, 0.0)));
    vec3 basis_v = cross(tilt, basis_u);
    float aperture = focal_length / f_stop;
    // Image plane values (precomputable)
    float image_plane = focus_a.z * focal_length /
        (dot(focus_a, tilt) - focal_length);
    vec2 shift = middle.xy / middle.z * -image_plane;

    // Image plane values (render-time)
    vec3 sensor = vec3(screen * 0.5 * sensor_size + shift,
                       -image_plane);
```

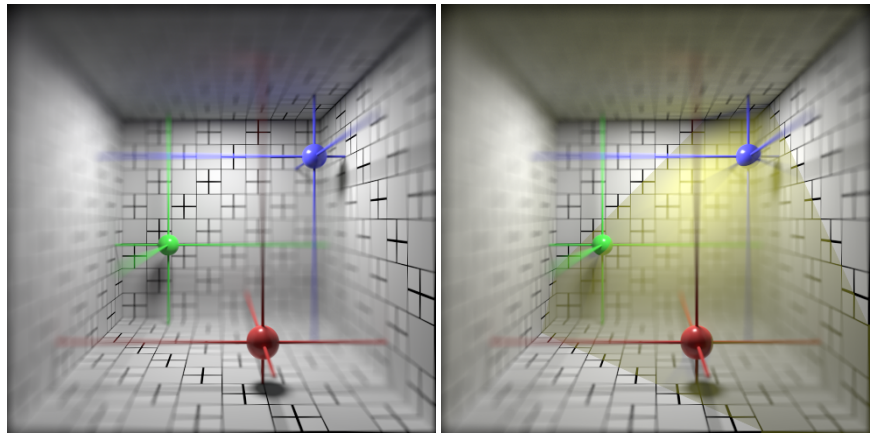
```
    // Lens values (render-time)
    float theta = 6.28318531 * random.x;
    float r = 0.5 * aperture * sqrt(random.y);
    vec3 lens = (cos(theta) * basis_u +
                 sin(theta) * basis_v) * r;
    // Focal plane values (render-time)
    vec3 focused = sensor * focal_length /
        (focal_length + dot(sensor, tilt));
    float flip = sign(dot(tilt, focused));

    ray_origin = lens;
    ray_direction = flip * normalize(focused - lens);
}
```

31.6 RESULTS

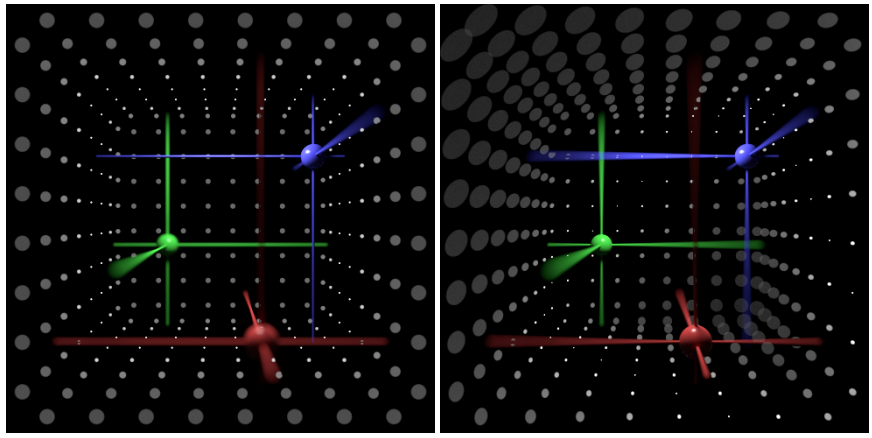
Figure 31-11 shows a test scene with the tilted lens in action and where the lens orientation and sensor distance has been solved for using the centers of the three spheres as our three points. All three points, despite being at different distances from the camera, are in perfect focus. Moreover, the focal plane intersects the box and produces a sharp appearance along diagonals in the grid texture. Although this is a relatively simple scene, already the defocus is much more visually complex than a simple blur masked along a gradient.

With the computer solving the tilt for us on each frame, we can easily animate it. For example, we can keep any three subjects in focus as they move about while the rest of the scene is de-emphasized and out of focus. Alternatively, we could hold the focal plane fixed in world space while moving the camera around. There are many possibilities for rendering with tilt-shift effects.



(a) *Three points defining the focus* **(b)** *Focal plane highlighted in yellow*

Figure 31-11. *Focus aligned to the plane through the spheres. (a) Without the plane of focus shown. (b) With the plane of focus highlighted in yellow.*



(a) *Standard lens* **(b)** *Tilted lens*

Figure 31-12. *Bokeh. (a) Circular bokeh with a standard thin lens focused on the blue sphere. (b) Elliptical bokeh with a tilted lens focused on all three spheres.*

Another interesting effect predicted by our tilt-shift model pertains to bokeh. Figure 31-12a shows a scene similar to the previous one except with the grid texture of the box replaced with very small, bright points and rendered with the non-tilted lens of the standard thin lens camera model focused on the blue sphere at middle distance. In this case, the small bright points of the surrounding box are circular bokeh, as usual.

However, if we return to the lens tilt from before with the plane of focus passing through the center of the three spheres, then some of the bokeh elongate into ellipses (Figure 31-12b). This should make some intuitive sense: from the perspective of the light cones from each bright point passing through the lens, the image sensor now cuts through at an oblique angle. In other words, the bokeh form conic sections.

The tilt-shift model presented in this chapter is incorporated in the production camera projection plugin included with Pixar's RenderMan. Though all of the rendered figures shown in this chapter were rendered with RenderMan, a complete self-contained demonstration that incorporates the sample code listings for the `thin_lens()` and `tilt_shift()` functions and recreates the scenes from Figure 31-11a and Figure 31-12 can be found online on Shadertoy at https://www.shadertoy.com/view/tlcBzN and at the book's source code website.

ACKNOWLEDGMENTS

Thanks to the RenderMan team at Pixar Animation Studios for supporting this work. It was done prior to the author joining Amazon. The city scene uses artwork assets from the "Mini Mike's Metro Minis" collection by Mike Judge, used under the Creative Commons Attribution 4.0 International license.

REFERENCES

[1] Cook, R. L., Porter, T., and Carpenter, L. Distributed ray tracing. *SIGGRAPH Computer Graphics*, 18(3):137–145, 1984.

[2] Merklinger, H. M. *Focusing the View Camera*. v. 1.6.1 first internet edition, 2010. http://www.trenholm.org/hmmerk/FVC161.pdf.

[3] Potmesil, M. and Chakravarty, I. A lens and aperture camera model for synthetic image generation. *SIGGRAPH Computer Graphics*, 15(3):297–305, 1981.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.


Amarillo



PART V
INTERSECTION

PART V

INTERSECTION

The idea of tracing ideal rays to determine object placement on a projection plane is famously credited to the sixteenth-century painter Albrecht Dürer, preceding by a few hundreds of years the ideation of computing machines. Even in the realm of three-dimensional computer graphics, ray tracing appeared early, with Arthur Appel's paper on the visual simulation of three-dimensional objects dating back to 1968. Needles to say, the mathematics of ray/object intersection is, by now, well researched.

Yet, the ever-increasing processing power at our disposal and the advent of practical, hardware-accelerated real-time ray tracing provide a novel context that invites us to revisit existing algorithms and allows the invention of new ones that push the complexity of our simulated three-dimensional worlds further. This part has contributions that address both cases.

Chapter 32, *Fast and Robust Ray/OBB Intersection Using the Lorentz Transformation*, and Chapter 33, *Real-Time Rendering of Complex Fractals*, are tutorials explaining the theory and practice of two important categories of ray/object intersections.

Chapter 34, *Improving Numerical Precision in Intersection Programs*, describes a widely applicable technique for improving numerical stability in ray/primitive intersection tests. The technique is not new, but it is not well described in any literature and thus not known to many practitioners.

Chapter 35, *Ray Tracing of Blobbies*, presents a novel way to intersect a widely used class of implicit surfaces, leveraging their local support to derive intervals that can be used to efficiently cull the intersection computations.

Chapter 36, *Curved Ray Traversal*, provides both an introduction to the ideas of gradient field tomography and a practical algorithm to trace rays traversing volumes with spatially varying refractive indices. This can be used both for visualization purposes and to perform reconstruction of compressible flows from acquired data.

Chapter 37, *Ray-Tracing Small Voxel Scenes*, compares the memory and performance requirements of different ways to implement voxel traversal on GPUs that provide hardware-accelerated ray tracing.

Angelo Pesce

CHAPTER 32

FAST AND ROBUST RAY/OBB INTERSECTION USING THE LORENTZ TRANSFORMATION

Rodolfo Sabino, Creto Augusto Vidal, Joaquim Bento Cavalcante-Neto, and José Gilvan Rodrigues Maia
Universidade Federal do Ceará (UFC)

ABSTRACT

We provide a numerical approach to compute oriented bounding boxes (OBBs) and a robust method to perform ray/OBB intersection based on the Lorentz transform. We also show how to compute additional intersection information (normal, face ID, and UV coordinates). Adopting OBBs instead of AABBs in a bounding volume hierarchy results in a significant reduction of traversal steps.

32.1 INTRODUCTION

Axis-aligned bounding boxes (AABBs) are the standard for bounding volume (BV) representation. They come from a history of research and industry-proven solutions and are backed by efficient, hardware-accelerated ray intersection algorithms. But try as one might, AABBs may not be tight-fitting BVs in the general case. By playing some simple tricks with ray transformation, we can literally spin them around and make AABBs better, at arguably negligible cost.

The tighter-fitting *oriented bounding box* (OBB) is encoded by a transformation matrix, which is applied to a ray before computing the intersection. This encoding can hold a wide variety of linear operations. The matrix can also be incorporated to modern ray tracing pipelines with little to no changes to underlying algorithms. Moreover, this solution can be performed in a parallel, branchless intersection test.

This methodology can be useful for many purposes. Here, we suggest exploring the properties of ray transformations for its use in OBB bounding volume hierarchies (BVHs; see Figure 32-1). For this purpose, this chapter

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_32



Figure 32-1. *Breakfast Room (left) rendered using a OBB BVH built on top of the proposed method. OBBs offer a better alternative to AABBs, as the former can offer a tighter fitting around objects (bottom right) and can reduce BVH traversal costs.*

provides the basic recipe to get the core of this idea working, namely how to build the BVs of the BVH: we suggest Principal Component Analysis (PCA) and BVH traversal, for which we go through the transformed ray/OBB intersection.

32.2 DEFINITIONS

A *ray* [7] is a line in 3D space that starts off from one point O and goes indefinitely toward one direction **d**. Testing ray/object intersection amounts to plugging the ray parametric equation $P(t)$ into the object's description and solving for t. Ray tracing is about using data from these intersection tests to feed shading algorithms. Listing 32-1 contains the typical ray payload data that we will compute in this chapter.

Bounding boxes are widely adopted in acceleration structures for rapid scene traversal and primitive culling. Bounding boxes are used as an inexpensive

Listing 32-1. *Intersection data return structure: Intersection point parameterized by t, normal* **n**, *face* **f**, *and texture coordinates* **uv**; `sNone` *denotes a miss.*

```
struct SurfaceHit
{
    float t;
    vec3  n;
    int   f;
    vec2 uv;
};
const SurfaceHit sNone = SurfaceHit(-1.0,vec3(0.0),0,vec2(0.0));
```

way of potentially discarding an intersection query before having to resort to costly object intersections: the smaller the box can be while still containing the object, the better. AABBs can't rotate, whereas OBBs can assume arbitrary orientation in 3D space. Consequently, OBBs usually outperform AABBs in terms of culling efficiency. On the other hand, building the AABB for an object is exact, simple, and fast: we just need to find the maximal and minimal coordinates belonging to the object's surface. Computing the optimal OBB takes $O(n^3)$. Algorithms that use PCA run in linear time and provide a decent approximation.

Transformations are applied to meshes' geometry, instancing them in different spatial configurations. Given a 3D input in homogeneous coordinates, 4×4 matrices can encode multiple types of linear transformations, such as scaling, rotation, reflection, shear, and translation. Multiple transformations can be efficiently represented by the result of matrix multiplications. Points and direction vectors operate differently in homogeneous coordinates. A point receives an additional $w = 1$ coordinate so the fourth columns of the matrices affect these positions, whereas directions have $w = 0$ so they are affected only by the 3×3 matrix core.

The *Lorentz transformation* is a term coined in physics literature and deals with linear transformations between coordinate frames. It is usually applied to positions and velocities, which are points and vectors, respectively. We use this mathematical background to represent rays in a coordinate frame–agnostic manner. This is similar to the way graphics pipelines process geometry when we use model-to-world matrices to instantiate geometry in a scene.

By carefully crafting an OBB transformation matrix, a ray from model space is converted into a reference, normalized AABB space (see Figure 32-2). All boxes in AABB space share precisely the same format, so specialized intersection tests can be developed. Moreover, the computed results are still valid in model and world space.

32.3 RAY/AABB INTERSECTION

For a more comprehensive view of the algorithm, see Chapter 2. An AABB can be seen as the intersection of six axis-aligned hyperplanes. The intersection between a ray and a plane given by a point P and normal $\mathbf{n}$ is

$$t = \frac{(P - O) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}. \tag{32.1}$$

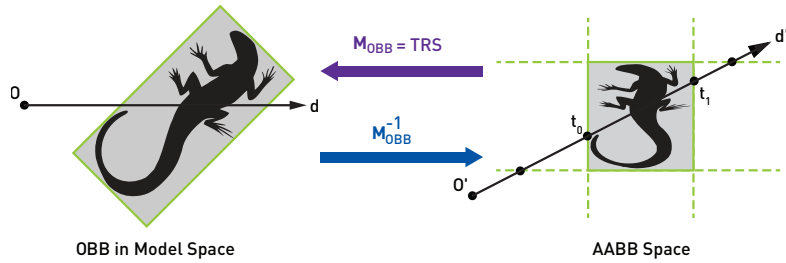


Figure 32-2. *Ray transformation between coordinate frames (blue and purple colored arrows). Rays in model space are mapped by the OBB inverse matrix $\mathbf{M}_{OBB}^{-1}$ into AABB space where the actual intersection test takes place.*

Half of the planes share each P_{min} and P_{max} points, and normal vectors are aligned with the $\pm X$, $\pm Y$, and $\pm Z$ axes. Therefore, the dot product operates only on the nonzero coordinate axis of the normal in Equation 32.1. For example,

$$\frac{(P_{\text{min}} - O) \cdot [1, 0, 0]}{\mathbf{d} \cdot [1, 0, 0]} = \frac{(P_{\text{min}} - O.x)}{\mathbf{d}.x}, \quad \text{for } \mathbf{n} = +X. \tag{32.2}$$

Thus, the intersection of all six planes can be vectored in the form

$$t_{\text{min}} = \frac{P_{\text{min}} - O}{\mathbf{d}}, \quad t_{\text{max}} = \frac{P_{\text{max}} - O}{\mathbf{d}}. \tag{32.3}$$

SIMD hardware capabilities are suitable to accelerate the *slabs algorithm* [1, 9] used for ray/AABB intersection.

Given t_{min} and t_{max}, the t's relative to each pair of parallel planes are ordered into two sets. One set s_c has planes that are closer to the ray, and the other set s_f has planes that are farther. Finally, s_c and s_f are checked for any overlapping.

An intersection exists when there is no overlapping between the closest and farthest set of t's, in other words, $t_0 \leq t_1$. The algorithm can be outlined as follows:

$$\begin{aligned} s_c &= min(t_{\text{min}}, t_{\text{max}}), \\ s_f &= max(t_{\text{min}}, t_{\text{max}}), \\ t_0 &= max(max(s_c.x, s_c.y), s_c.z), \\ t_1 &= min(min(s_f.x, s_f.y), s_f.z), \end{aligned} \tag{32.4}$$

Listing 32-2. *Ray/AABB intersection function based on the slabs algorithm [1, 9].*

```
SurfaceHit intersectRayAABB(vec3 ro,vec3 rd,vec3 pmin,vec3 pmax)
{
    // Basic ray/AABB intersection (see Section 32.3)
    vec3  tmin = (pmin-ro)/rd;
    vec3  tmax = (pmax-ro)/rd;
    vec3  sc   = min(tmin,tmax);
    vec3  sf   = max(tmin,tmax);
    float t0   = max(max(sc.x,sc.y),sc.z);
    float t1   = min(min(sf.x,sf.y),sf.z);
    if(!(t0 <= t1 && t1 > 0.0)) return sNone;

    // Computing additional intersection data (see Section 32.5)
    // Normals
    vec3 n = -sign(rd)*step(sc.yzx,sc.xyz)*step(sc.zxy,sc.xyz);

    // Face ID
    int f  = int(dot(abs(n),vec3(1,2,4)));
        f ^= int(any(greaterThan(n,vec3(0))));

    // Texture coordinates
    vec3 uv3 = ro+t0*rd;
    vec2 uv2 = 0.5+mix(uv3.xy,uv3.zz,abs(n.xy));
         uv2 = mix(uv2,vec2(1.0-uv2.x,uv2.y),max(n.x,max(n.y,-n.z)));

    return SurfaceHit(t0,n,f,uv2);
}
```

where t_0 gives the signed distance between the ray origin and the point of intersection in the AABB, if an intersection does occur (Listing 32-2).

Notice that this algorithm relies on standard floating-point behavior. The result of the division by zero gives the appropriate signed infinity. On the other hand, only very specific cases produce `NaN` (not a number) due to the computation of `tmin`, `tmax` = 0/0. This typically occurs when the ray is tangent to the box's surface. As `NaN` propagates through `min()` and `max()`, such rays are properly discarded by checking consistent t_0 and t_1 values (see line 10 in Listing 32-2).

32.4 RAY/OBB INTERSECTION

Depending on how one handles the problem of ray/OBB intersection, the algorithm can become quite complex. Luckily, we can use the properties of ray transformations to simplify our intersection. The key idea is to imagine the OBB as an instance of a standard, "unit" AABB by means of a matrix $\mathbf{M}_{\text{OBB}}$ (see Figure 32-2).

We use $\mathbf{M}_{\text{OBB}}^{-1}$ to transform the ray into AABB space. We define this standard AABB space as a unit cube centered at the origin:

$$P_{\text{min}} = [-0.5, -0.5, -0.5], P_{\text{max}} = [0.5, 0.5, 0.5]. \tag{32.5}$$

You need to generate an invertible matrix representing this transformation from the unit AABB into your model's OBB. Preferably, you also should adopt a robust algorithm that can efficiently compute the inverse [3].

We give an example of computing this matrix using a PCA-based technique over a triangular mesh, but our technique is easily adaptable to other analytical solutions that are suitable for the model representation you have at hand. The code for this technique as described in the Equation 32.6 below is available on Shadertoy [5].

For its use as a bounding box, this matrix can be precomputed offline for a mesh's vertex set $\mathbb{P}$. We derive the matrix $\mathbf{M}_{\text{OBB}} = \mathbf{TRS}$ as the combination of the scale **S**, rotation **R**, and translation **T** parts extracted from the mesh by PCA. The rotational part **R** is derived from the eigenvectors of the covariance matrix for $\mathbb{P}$. Scaling can be obtained by computing the minimal and maximal coordinates from the mesh points after being transformed by the inverse of the rotation matrix. Finally, translation is computed taking into consideration the sample mean $\overline{P}$ computed by the covariance matrix and the centroid V'_{center} of the transformed mesh, as follows:

$$\begin{aligned}
\mathbb{P}' &= \bigcup_{i=1}^{n} \mathbf{R}^{\top}(P_i - \overline{P}), \\
V'_{\text{min}} &= min(\mathbb{P}'), \\
V'_{\text{max}} &= max(\mathbb{P}'), \\
V'_{\text{center}} &= (V'_{\text{min}} + V'_{\text{max}}) \times 0.5, \\
\mathbf{S} &= \text{scale}(V'_{\text{max}} - V'_{\text{min}}), \\
\mathbf{T} &= \overline{P} + \mathbf{R}V'_{\text{center}}, \\
\mathbf{M}_{\text{OBB}} &= \mathbf{TRS}.
\end{aligned} \tag{32.6}$$

This method rotates the vertices by the eigenvectors basis, then computes an AABB for the vertices in this reference frame. We can be sure to compute a tight-fitting OBB by this method, but it may not yield the smallest box configuration [2]. Being a statistical method, PCA relies on spatial patterns of the geometry to figure out the axis of the OBB. The eigenvector basis follows the density of vertices aligned with the eigenvectors. If you have a significant

portion of approximately aligned vertices in your mesh, when creating the OBB the rotational part will use this diagonal on its basis. This problem can be mitigated by ignoring vertices outside the convex hull.

The matrix $\mathbf{M}_{\text{OBB}}$ encodes the transformation of the unit AABB into the mesh's local OBB. So, $\mathbf{M}_{\text{OBB}}^{-1}$ transforms a ray in model space into that mesh's unit AABB space (see Figure 32-2). Further transformations can also be encoded on this matrix. Given the model's instance matrix $\mathbf{M}_{\text{inst}}$, we can transform the ray from world space into that model's unit AABB space by using $\mathbf{M}_{\text{AABB}} = \mathbf{M}_{\text{OBB}}^{-1}\mathbf{M}_{\text{inst}}^{-1}$:

$$\begin{aligned} O'' &= \mathbf{M}_{\text{AABB}}O, \\ \mathbf{d}'' &= \mathbf{M}_{\text{AABB}}\mathbf{d}. \end{aligned} \tag{32.7}$$

Finally, we plug the new ray $P''(t) = O'' + t\mathbf{d}''$ into the ray/AABB intersection algorithm as follows:

$$\texttt{intersectRayAABB}\left(O'', \mathbf{d}'', [-0.5, -0.5, -0.5], [0.5, 0.5, 0.5]\right) \tag{32.8}$$

The resulting t from this intersection is valid in world space, but normals may need to be transformed by $(\mathbf{M}_{AABB}^{-1})^T$ to be in world space as well.

32.5 COMPUTING ADDITIONAL INTERSECTION DATA

The ray/OBB intersection algorithm can be used both for BV queries and for intersecting box primitives. In the first use case, we are only interested in whether an intersection happens or not, as then only the intersection conditionals are necessary (see line 10 in Listing 32-2). For the sake of completeness, we created this section that covers the second use case. Here, we show how to compute additional surface hit data that can be used for shading the primitive.

The ray/AABB intersection algorithm inherently computes the intersection t's for all six faces. These values are used to compute intersection data (see Listing 32-1).

Normals of an AABB have a component set to ± 1 in exactly one axis and zero in the other axes. The signs of the visible normals are computed using the sign from $-\mathbf{d}$ because our unit AABB is centered at the origin. The signed unit component goes to the respective axis of the face where the smallest t belongs (see line 14 of Listing 32-2). These normals are in AABB space, so

Listing 32-3. *Ray/OBB intersection function based on the Lorentz transformation.*

```
1  SurfaceHit intersectRayOBB(vec3 ro,vec3 rd,mat4 m)
2  {
3      // invm can be precomputed.
4      mat4 invm = inverse(m);
5
6      // normal matrix used to transform AABB normals into OBB coordinates
7      mat4 nm = transpose(invm);
8
9      // ray transformation; rd does not need to be normalized.
10     vec3 roPrime = (invm*vec4(ro,1)).xyz;
11     vec3 rdPrime = (invm*vec4(rd,0)).xyz;
12
13     SurfaceHit s = intersectRayAABB(roPrime,rdPrime,vec3(-0.5),vec3(0.5));
14     if(s.t < 0.0) return s;
15
16     s.n = normalize((nm*vec4(s.n,0)).xyz);
17     return s;
18 }
```

they need to be transformed by the normal matrix $\mathbf{N} = (\mathbf{M}'^{-1}_{\text{OBB}})^T$ to be placed in model space (see line 16 of Listing 32-3).

Having the normal, determining the *face ID* is straightforward: we use a 3-bit integer to store the face ID. The first, second, and third bits are turned on for X, Y, and Z aligned normals, respectively. We xor +1 when the nonzero coordinate is positive and 0 otherwise (see line 18 of Listing 32-2).

The *UV coordinates* are encoded on the intersection point because the AABB is a unit cube centered at the origin. We determine which coordinate to drop based on the normal and then translate the remaining coordinates so the result lies in the [0, 1] range. Finally, we flip the UV coordinates on opposite planes from each axis so that all of them have the same orientation from an outside view (see line 23 of Listing 32-2).

32.6 CONCLUSION

We present a convenient method for ray/OBB intersection based on ray transformation. It is built over the ray/AABB algorithm already available in ray tracing hardware and only adds the cost of two matrix/vector multiplications.

This method fits nicely on instance-based interactive graphics pipelines: it is robust and fast and has the potential to further optimize BVHs (see Figure 32-3). Wald et al. [8] provide a creative implementation for GPU-accelerated OBB traversal using a very similar approach. However,

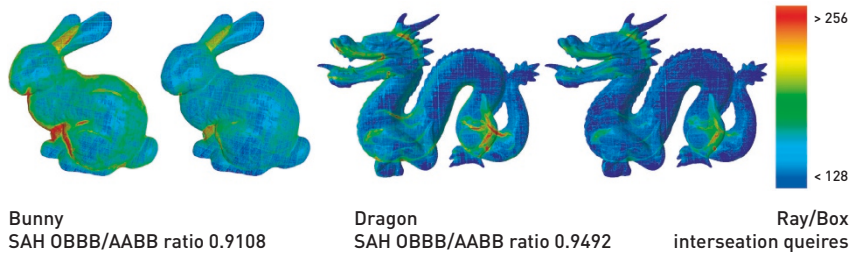


Figure 32-3. *Heatmap illustrating the ray BVH traversal cost for different models using AABBs (left) and OBBs (right). The gradient represents the number of intersection queries between a ray and BVs required during ray casting until the ray hits the geometry at the bottom of the tree. The Surface Area Heuristic (SAH) ratio between both types of BVHs can be used to argue that BVHs of smaller SAH lead to reduced traversal costs.*

Majercik et al. [4] provide the current state-of-the-art implementation of similar methods. Both methods are a step ahead for future hardware OBB BVH builders.

Furthermore, this method does not hinder the inventive use of matrices in the next generation of graphics technology, as it supports a wide variety of transformations, such as shear, mirroring, and nonuniform scaling, and use of the homogeneous coordinate for scaling (see Figure 32-4).

We provide an implementation code for building the OBB from a mesh based on the method we described in Equation 32.6 at Shadertoy [5]. We also provide an implementation for ray/AABB intersection (Listing 32-2), ray/OBB intersection (Listing 32-3), and the intersection data structure (Listing 32-1) [6].

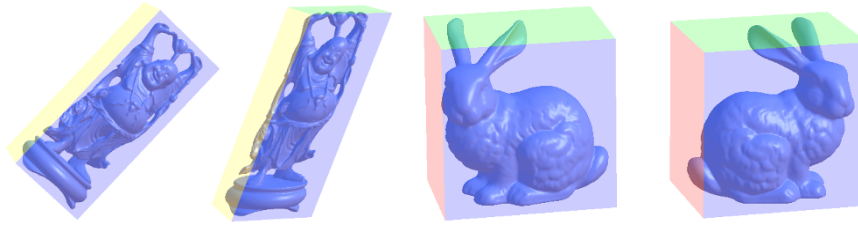


Figure 32-4. *Our implementation supports shearing, nonuniform scaling, and mirroring.*

REFERENCES

[1] Kay, T. L. and Kajiya, J. T. Ray tracing complex scenes. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 269–278, 1986. DOI: 10.1145/15922.15916.

[2] Larsson, T. and Källberg, L. Fast computation of tight-fitting oriented bounding boxes. In E. Langyel, editor, *Game Engine Gems 2*, pages 1–20. CRC Press, 2011.

[3] Maia, J. G. R., Vidal, C. A., and Cavalcante-Neto, J. B. Efficient collision detection using transformation semantics. In S. Jacobs, editor, *Game Programming Gems 7*, pages 179–189. Charles River Media, 2008.

[4] Majercik, A., Crassin, C., Shirley, P., and McGuire, M. A ray-box intersection algorithm and efficient dynamic voxel rendering. *Journal of Computer Graphics Techniques Vol*, 7(3), 2018.

[5] nolcip. OBB construction using PCA. https://www.shadertoy.com/view/sdsGDM, March 18, 2021.

[6] nolcip. Transformed ray/OBB. https://www.shadertoy.com/view/XtycDK, February 17, 2019.

[7] Shirley, P., Wald, I., Akenine-Möller, T., and Haines, E. What is a ray? In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 15–19. Apress, 2019.

[8] Wald, I., Morrical, N., Zellmann, S., Ma, L., Usher, W., Huang, T., and Pascucci, V. Using hardware ray transforms to accelerate ray/primitive intersections for long, thin primitive types. *Proceedings of the ACM on Computer Graphics and Interactive Techniques (Proceedings of High Performance Graphics)*, 3(2):17:1–17:16, 2020. DOI: 10.1145/3406179.

[9] Williams, A., Barrus, S., Morley, R. K., and Shirley, P. An efficient and robust ray-box intersection algorithm. In *ACM SIGGRAPH 2005 Courses*, 9–es, 2005. DOI: 10.1145/1198555.1198748.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 33

REAL-TIME RENDERING OF COMPLEX FRACTALS

Vinícius da Silva,[1] Tiago Novello,[2] Hélio Lopes,[1] and Luiz Velho[2]
[1]Pontifical Catholic University of Rio de Janeiro
[2]National Institute for Pure and Applied Mathematics

ABSTRACT

This chapter describes how to use intersection and closest-hit shaders to implement real-time visualizations of complex fractals using distance functions. The Mandelbulb and Julia sets are used as examples.

33.1 OVERVIEW

Complex dynamics fractals have interesting patterns that can be used to create special effects or moods in 3D scenes. Walt Disney Animation Studios used a Mandelbulb in *Big Hero 6* [12] to design the inside of a wormhole. Marvel Studios also used Mandelbulbs to produce the magical mystery tour scene in *Doctor Strange* [20] and other 3D fractals in *Guardians of the Galaxy Vol. 2* [6]. *Lucy* [13] and *Annihilation* [7] have other examples of fractal-based effects. Most of them do not intersect the fractal directly, but convert it to point clouds or VDB volumes [14]. Using DirectX Raytracing (DXR) via custom intersection shaders allows the integration of fractals with triangle-based scenes and path tracing automatically, with hardware acceleration.

We are interested in rendering 3D *complex fractals* associated with polynomials of the form $f(Z) = Z^k + C$. For this, we must consider a domain for f that allows multiplication, so Z^k can be evaluated. The complex numbers and quaternions do so, producing 2D and 4D fractals. However, by the Frobenius theorem [1], there is no similar operation for 3D points, so there are no natural 3D complex fractals. To overcome this, we can work with quaternions and take 3D slices of the resulting 4D fractals, or we can consider an informal multiplication of 3D points.

Julia sets and the Mandelbrot set [5] are popular examples of complex fractals. As such, they can be rendered in 3D using both approaches just

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_33

described. For the informal multiplication in 3D space, these fractals are called *Juliabulb* and *Mandelbulb*, respectively [16]. To cover both approaches, this chapter focuses on the rendering of Julia sets using 3D slices of the 4D fractal and the Mandelbulb.

33.1.1 JULIA SETS

Examples of *Julia sets* arise from the exploration of the convergence of the sequence given by the iterations of the quadratic function $f(Z) = Z^2 + C$. Specifically, a (filled-in) Julia set consists of the set of points Z in the complexes/quaternions, where the sequence $f^n(Z)$ has a finite limit. The expression f^n means that f is composed n times. Changing the constant C produces different Julia sets.

Using the complex plane as the domain of the quadratic function f results in the traditional images of 2D Julia sets. Norton [15] extended this class of fractals to 4D considering that the quaternions $\mathbb{Q}$ are the domain of f. We denote such fractals as *4D Julia sets* or *quaternion Julia sets* interchangeably. Three-dimensional slices of a quaternion Julia set can be rendered by letting its real part be equal to 0, i.e., by restricting the fractal to $\{ai + bj + ck + d \in \mathbb{Q} | \, d = 0\}$.

To visualize a 2D Julia set, we check whether a point on the complex plane diverges. Thus, it suffices to compute the sequence $f^n(Z)$ and see how quickly its magnitude increases. This test can be applied to points (pixels) in an image, resulting in an illustration of a 2D Julia set. Figure 33-1 shows the 2D Julia set in the orange region, where the iterations of its points by $f(Z) = Z^2 + C$ have finite limit. The circle points illustrate the iterations (also called *orbits*) of the square points. As expected, the blue sequence diverges and the green one converges.

This approach is inefficient to render 3D slices of a quaternion Julia set. However, as we are interested in the fractal "surface," ray tracing is an appropriate technique. Ray tracing fractals dates back to the work of Hart et al. [11], which uses a distance estimator (described in [15]) to speed up the ray tracing process. Recently, Quilez [17] presented real-time visualizations of those 3D slices using pixel shaders, applying techniques similar to those defined in [11].

Inspired by the work of Quilez, we use DXR shaders to render 3D slices of quaternion Julia sets. Figure 33-2 shows one such slice, cut by a plane in 3D.

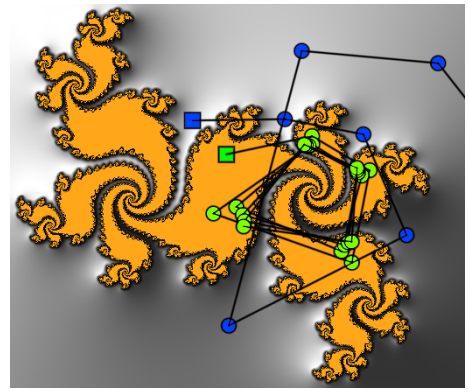


Figure 33-1. *The 2D Julia set (orange) associated with the map $f(Z) = Z^2 + C$, with $C = 0.28 - i \cdot 0.49$. The green (blue) sequence of dots represents $f^n(Z)$, where Z is the green (blue) square. The green (blue) sequence converges (diverges) because the green (blue) square is inside (outside) of the Julia set. (Image generated using a modified version of Iñigo Quilez's Complex Dynamics shader in Shadertoy [18].)*

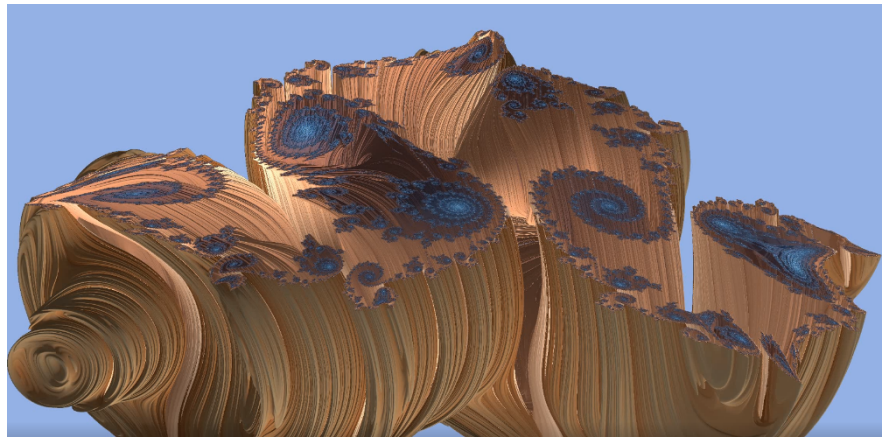


Figure 33-2. *A 3D slice of a quaternion Julia set, cut by a plane in 3D. The image restricted to the plane looks like a 2D Julia set. (Rendered using the shaders in this chapter.)*

33.1.2 MANDELBULB

The *Mandelbrot set* associated with a polynomial $f(Z) = Z^k + C$ is the set of points C such that the sequence $f^n(0)$ (the orbit of the zero element) has finite limit. Therefore, it serves as an interface for the constants C used in the Julia set definition. That is, each point in the Mandelbrot set corresponds to a Julia set.

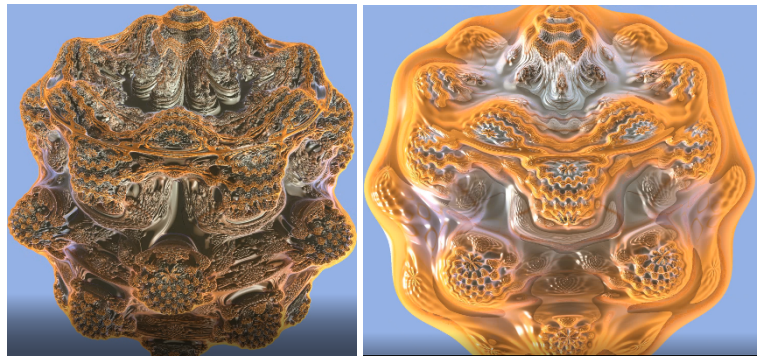


Figure 33-3. *Two frames of an animated Mandelbulb rendered using the shaders described in this chapter, considering $f(P) = P^8 + C$. The animation is created by varying the number of steps in the ray marching over time.*

The *Mandelbulb* is a fractal commonly used to represent the Mandelbrot set in three dimensions. It was constructed by White [21] and Nylander [16]. They defined an informal kth power operation for 3D points to evaluate $f(P) = P^k + C$ in the 3D space. This is based on informally extending the multiplication (related to 2D rotations) of complex numbers to 3D space. In this context, each Julia set associated with a point C in the Mandelbulb is a *Juliabulb*.

The formula for the kth power of a point $P \in \mathbb{R}^3$ is

$$P^k := r^k \left(\sin(k\theta)\cos(k\phi), \sin(k\theta)\sin(k\phi), \cos(k\theta)\right), \tag{33.1}$$

where $r = |P|$ is the norm of P and $\theta = \arctan(P_y/P_x)$ and $\phi = |(P_x, P_y)|/P_z$ are the spherical coordinates of $P/|P|$. We give the motivation of this formula. Let $Z = r(\cos\theta + i\sin\theta) = re^{i\theta}$ be a complex number represented by Euler's formula. Then, the kth power of Z is $Z^k = r^k e^{i(k\theta)} = r^k(\cos(k\theta) + i\sin(k\theta))$.

Figure 33-3 shows the Mandelbulb associated with $f(P) = P^8 + C$, using Equation 33.1.

33.2 DISTANCE FUNCTIONS

In this section, we present an approximation of distance functions to complex fractals. We follow the definition and notations of Quilez [19], presenting a non-rigorous but intuitive discussion about it.

A distance function d can be used to render objects defined by its zero level set $\{P \in \mathbb{R}^3 | \, d(P) = 0\}$. A well-known technique for this task is *sphere*

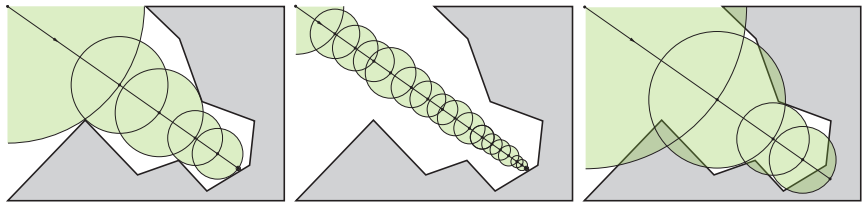


Figure 33-4. *Sphere tracing with different approximations for the distance function: a very close approximation (left), a function smaller than the distance (center), and a function greater than the distance (right). The center and right options are problematic because of performance and correctness, respectively.*

tracing [10, 11], which works as follows. Let P_0 and $\mathbf{v}$ be the ray origin and ray direction, respectively. The intersection between the ray $P_0 + t\mathbf{v}$ and the zero level set of d is approximated by iterating $P_{n+1} = P_n + \mathbf{v}\max\{d(P_n), \epsilon\}$. Figure 33-4 illustrates the algorithm using several approximations for d with the same zero level set. The original algorithm has been improved by recent works [2, 8].

Let $f(Z) = Z^k + C$ be a polynomial map. Our objective is to define a distance from the set of points Z, where the sequence $f^n(Z)$ has a limit. Remember that f^n means that f is composed n times, i.e., $f^n(Z) = (f^{n-1}(Z))^k + C$. The *Böttcher map* $\phi_C(Z) = \lim_{n\to\infty}(f^n(Z))^{k^{-n}}$ is used to derive the distance function. This map is a deformation of the underlying space.

We provide an informal discussion about the importance of the Böttcher map. Let Z be an element such that the sequence $f^n(Z)$ diverges, i.e., $\lim_{n\to\infty} |f^n(Z)| = \infty$. Thus, for an n big enough, $f^n(Z)$ is far away from the fractal. In this case, the term $f^n(Z)$ will dominate over C. Then, we can forget about C in the expression $f(Z) = Z^k + C$ to obtain $f^{n+1}(Z) \approx (f^n(Z))^k$. In this case, the expression of f turns out to be $f_0(Z) = Z^k$, so we can undo the interactions by considering $(f_0^n(Z))^{k^{-n}} = Z$. Therefore, $\phi_0(Z) = Z$ when $f^n(Z)$ diverges.

According to the aforementioned property of $\phi_0(Z)$, the Böttcher map approximates to the identity $\lim_{Z\to\infty}\phi_C(Z) = Z$ as we move away from the fractal. For example, Figure 33-5 shows this situation for the Julia set associated with $f(Z) = Z^2 + C$. The regions distant from the Julia set (blue square) receive less deformation, while the regions near it (green square) deform more.

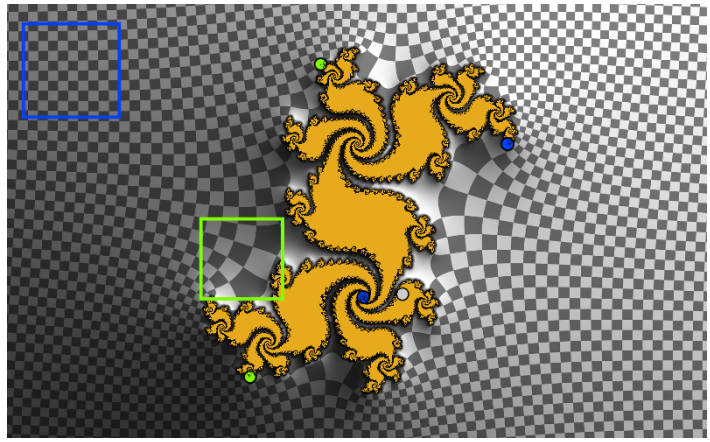


Figure 33-5. *Böttcher map of the Julia set. The blue square shows an area with small deformations, where $\phi_C(Z) \approx Z$. The green square shows an area near the Julia set, where deformations are stronger. The colored points can be mostly ignored for our purposes (the gray point is C, and the other ones are fixed points of the dynamical system). (Image generated using Iñigo Quilez's Complex Dynamics shader in Shadertoy [18].)*

Based on the Böttcher map, we now search for a function that approximates the distance to the fractal. For this, we define

$$G(Z) = \log|\phi_C(Z)| = \lim_{n\to\infty} \frac{\log|f^n(Z)|}{k^n}. \tag{33.2}$$

It is easy to see that this function is smooth and is 0 at the fractal because $f^n(Z)$ converges and k^n grows exponentially. In addition, as we move away from the fractal, $G(Z)$ tends to $\log|Z|$ because $\lim_{Z\to\infty} \phi_C(Z) = Z$. Finally, because of its continuity, the function G gets close to zero as we get close to the set. Even though G has a few properties of the distance function (it has the same zero level set), it is not a good approximation as it tends to be $\log|Z|$ far away from the fractal. In other words, it tends to have the performance problems shown in Figure 33-4 (center).

To find a better approximation of the distance, we use the first-order Taylor expansion of G. The result is an upper bound estimation $d(Z) = |G(Z)|/|\nabla G(Z)|$, which also uses the gradient of G. Next, we explain this estimator in detail.

Let Z be a point and $\mathbf{v}$ be a vector such that $Z + \mathbf{v}$ is the closest point in the fractal from Z, i.e., $|\mathbf{v}|$ is the desired distance. Then, we have that $G(Z + \mathbf{v}) = 0$, so using the Taylor expansion of G, we obtain $0 = G(Z) + \nabla G(Z) \cdot \mathbf{v} + O(|\mathbf{v}|^2)$. Let's assume that the quadratic term is negligible, i.e., $0 = |G(Z) + \nabla G(Z) \cdot \mathbf{v}|$. Then,

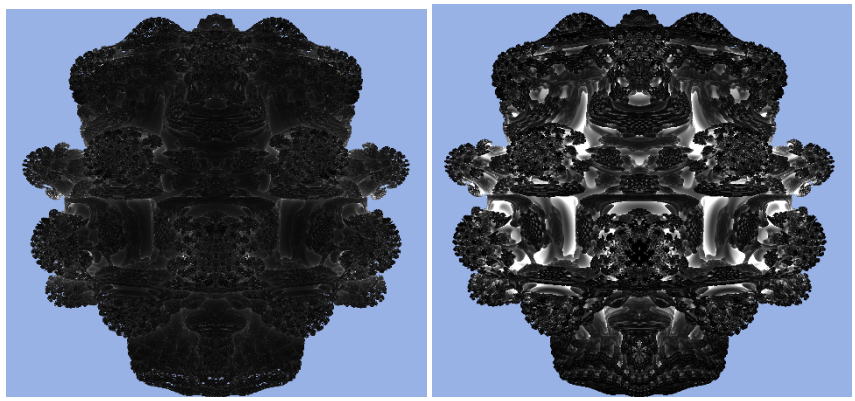


Figure 33-6. *Rendering the sphere tracing iterations based on the distance function approximation: Equation 33.3 (left) and Equation 33.2 (right). The color ranges from black (less iterations) to white (more iterations). As expected, Equation 33.3 performs better.*

using the inequalities $|G(Z) + \nabla G(Z) \cdot \mathbf{v}| \geq |G(Z)| - |\nabla G(Z) \cdot \mathbf{v}|$ and $|\nabla G(Z) \cdot \mathbf{v}| \leq |\nabla G(Z)||\mathbf{v}|$, we get an upper bound to the distance $|\mathbf{v}| \geq |G(Z)|/|\nabla G(Z)|$. We derive the first inequality from the *triangle inequality* and the second comes from the *Cauchy–Schwarz inequality*.

Using the definition of $G(Z)$ (Equation 33.2) and its derivatives $\nabla G(Z) = |(f^n)'(Z)|/k^n|f^n(Z)|$, the distance function $d(Z) = |G(Z)|/|\nabla G(Z)|$ can be expressed as

$$d(Z) = \lim_{n\to\infty} \frac{|f^n(Z)| \log |f^n(Z)|}{|(f^n)'(Z)|}. \tag{33.3}$$

Figure 33-6 shows the sphere tracing performance using the distance approximations given by Equations 33.2 and 33.3. Performance is better using Equation 33.3.

From Equation 33.3, we have to compute both f^n and its derivative during the iteration loop. We compute the sequences using

$$f^{n+1} = (f^n)^k + C, \tag{33.4}$$

$$(f^{n+1})' = k \cdot (f^n)^{k-1} \cdot (f^n)'. \tag{33.5}$$

Equation 33.4 is the recursion corresponding to the iteration of f, and its derivative is presented in Equation 33.5 with the initial condition $(f^0)' = 1$, because $f'(Z) = k \cdot Z^{k-1}$. Iterating these equations results in an algorithm (see Listing 33-4) to compute the distance function in Equation 33.3.

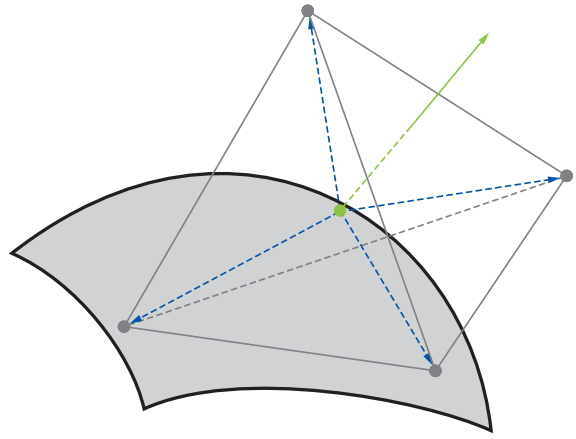


Figure 33-7. *Gradient approximation (green arrow) at the point P (green dot). The gray dots are the tetrahedron vertices $P + h\mathbf{v}_i$ and the dashed blue vectors are $\mathbf{v}_i$, whose weighted average approximates $\nabla f(P)$. The weight for each vertex is $f(P + h\mathbf{v}_i)$.*

To do the shading of the point Z of the fractal, we need an approximation of its normal vector. As the function $d(Z) = G(Z)/|\nabla G(Z)|$ is an approximation of the distance function from the fractal, its gradient provides the desired normal. We could compute the gradient analytically using

$$\nabla d = \frac{\nabla G|\nabla G| - G\nabla|\nabla G|}{|\nabla G|^2}, \tag{33.6}$$

which comes from the quotient rule of the gradient. When restricted to the fractal, the expression turns out to be $\nabla d = \nabla G/|\nabla G|$ because the function G is 0 at the set. Instead, we consider a simple numerical approach for convenience.

We use an efficient procedure to numerically compute the gradient ∇f of a function $f : \mathbb{R}^3 \to \mathbb{R}$. Let $\mathbf{v}_0 = (1, -1, -1)$, $\mathbf{v}_1 = (-1, -1, 1)$, $\mathbf{v}_2 = (-1, 1, -1)$, and $\mathbf{v}_3 = (1, 1, 1)$ be the vertices of a tetrahedron; $\mathbf{v}_i$ is a subset of the vertices of the cube $[-1, 1]^3$, where $[-1, 1]$ is an interval. The gradient $\nabla f(P)$ can be approximated by

$$\nabla f(P) \approx \frac{1}{4h} \sum_i \mathbf{v}_i f(P + h\mathbf{v}_i). \tag{33.7}$$

Figure 33-7 gives a geometrical intuition of Equation 33.7. To derive that expression, we define $\mathbf{m} := \sum_i \mathbf{v}_i f(P + h\mathbf{v}_i)$ and rewrite it using $\sum_i \mathbf{v}_i = (0, 0, 0)$ to obtain $\mathbf{m} = \sum_i \mathbf{v}_i (f(P + h\mathbf{v}_i) - f(P))$. Using $(f(P + h\mathbf{v}_i) - f(P))/h$ as an approximation for the derivative $\frac{\partial}{\partial \mathbf{v}_i} f(P)$ of f in the direction $\mathbf{v}_i$, we get $\mathbf{m} \approx \sum_i \mathbf{v}_i h \frac{\partial}{\partial \mathbf{v}_i} f(P)$. From

calculus, we have $\frac{\partial}{\partial \mathbf{v}_i} f(P) = \mathbf{v}_i \cdot \nabla f(P)$, therefore $\mathbf{m} \approx \sum_i \mathbf{v}_i h \mathbf{v}_i \cdot \nabla f(P)$. We look at the component x of $\mathbf{m}$; the computations are analogous for $\mathbf{m}_y$ and $\mathbf{m}_z$:

$$\frac{\mathbf{m}_x}{h} \approx \sum_i (\mathbf{v}_i)_x \mathbf{v}_i \cdot \nabla f(P) = \sum_i (\mathbf{v}_i)_x \mathbf{v}_i \cdot \nabla f(P) = (4, 0, 0) \cdot \nabla f(P) \tag{33.8}$$

We used the linearity of the dot product in the second equality. It is easy to verify that $\sum_i (\mathbf{v}_i)_x \mathbf{v}_i = (4, 0, 0)$, which explains the last equality. As a result, we have $\frac{\mathbf{m}}{4h} \approx \nabla f(P)$, as stated in Equation 33.7.

33.3 IMPLEMENTATION

We assume several procedural objects in the scene, each one with a matrix transforming world space into the local space of the axis-aligned bounding box (AABB). The shaders were implemented in HLSL, using the Proceduray engine [3]. The host setup is beyond the scope of this chapter, but we refer to [3] as an in-depth guide to do so.

33.3.1 JULIA SETS

Listing 33-1 shows the code for the intersection shader of a 3D slice of a quaternion Julia set. The function `GetRayInLocalSpace()`, in line 4, computes the origin and direction in the local coordinates of the underlying AABB. The corresponding ray is passed to `IntersectionJuliaTest()` (Listing 33-2), in line 8, which determines the ray parameter corresponding to the intersection between the ray and the Julia set and the normal at the hit point. Note that `thit` is a `float2` because it also contains the number of iterations in the distance function. Those values are used in the closest-hit shader (Listing 33-6) to compute the shading of the object.

Listing 33-1. *Intersection shader.*

```
1  [shader("intersection")]
2  void IntersectionJulia()
3  {
4      Ray ray = GetRayInLocalSpace();
5      float2 thit;
6      ProceduralPrimitiveAttributes attr;
7      float3 pos;
8      bool test=IntersectionJuliaTest(ray.ori,ray.dir,attr.normal,thit);
9      if (test && thit.x < RayTCurrent())
10     {
11         attr.normal = mul(attr.normal, (float3x3) WorldToObject3x4());
12         attr.color = float4(thit, 0.f, 0.f);
13         ReportHit(thit.x, /*hitKind*/ 0, attr);
14     }
15 }
```

`IntersectionJuliaTest()` (Listing 33-2) returns a boolean indicating if there is an intersection and outputs `resT` and `normal`. It calls functions for finding the distance using sphere tracing (Listing 33-3) in line 5 and to calculate the normals at the intersection point (Listing 33-5) in line 10.

Listing 33-2. *IntersectionJuliaTest.*

```
1  bool IntersectionJuliaTest(in float3 ro, in float3 rd,
2      inout float3 normal, inout float2 resT)
3  {
4      resT = 1e20;
5      float2 tn = JuliaSphereTracing(ro, rd);
6      bool cond = (tn.x >= 0.0);
7      if (cond)
8      {
9          float3 pos = (ro + (tn.x * rd));
10         normal = CalcNormal(pos);
11         resT = tn;
12     }
13     return cond;
14 }
```

`JuliaSphereTracing()` (Listing 33-3) computes an approximation of the first intersection. The algorithm delimits an intersection search interval at lines 3 and 4. In line 5, it updates that interval based on a bounding sphere and two clipping planes (to cut the Julia set, as in Figure 33-2). The sphere tracing loop (lines 11–19) uses function `Dist()` (Listing 33-4), in line 13, to calculate the distance.

Listing 33-3. *Julia set sphere tracing.*

```
1  float2 JuliaSphereTracing(in float3 ro, in float3 rd)
2  {
3      float tmin = kPrecis; // kPrecis = 0.00025f
4      float tmax = 7000.f;
5      if (!CheckBoundaries(ro, rd, tmin, tmax))
6          return float2(-2.0, 0.0);
7      float2 res = { -1.0, -1.0 };
8      float t = tmin;
9      float lt = { 0.0 };
10     float lh = { 0.0 };
11     for (int i = 0; i < 1024; i++)
12     {
13         res = Dist(ro + (rd * t));
14         if (res.x < kPrecis) break;
15         lt = t;
16         lh = res.x;
17         t += min(res.x, 0.2);
18         if (t > tmax) break;
19     }
20     res.x = (t < tmax) ? t : -1.0f;
21     return res;
22 }
```

`Dist()` (Listing 33-4) implements Equations 33.4 and 33.5, used to approximate the distance function given in Equation 33.3. The variable `z`, defined in line 3, is a `float4` representing a quaternion that is the initial condition of the recursion in Equation 33.4. The 3D slicing is done by letting the last coordinate of `z` be 0. Line 4 defines the initial condition of the recursion in Equation 33.5. The loop (lines 7–14) does the iterations of the system (Equations 33.4 and 33.5). There is a break, in line 12, to stop the loop when the sequence given by the iterations of `z` (line 10) diverges. Finally, line 15 computes an approximation of the distance function using Equation 33.3. This implementation is particular for the Julia set associated with $f(Z) = Z^3 + C$, where $C = (-2i + 6j + 15k - 6)/22 \in \mathbb{Q}$. Other Julia sets can be rendered by varying C and the exponent of Z.

Listing 33-4. *Julia set distance.*

```
1  float2 Dist(in float3 p)
2  {
3      float4 z = float4(p, 0.0); // 3D slicing
4      float dz2 = 1.0;
5      float m2 = 0.0;
6      float n = 0.0;
7      for (int i = 0; i < 200; i++)
8      {
9          dz2 *= 9.0 * QuatLength2(QSquare(z));
10         z = QuatCube(z) + kc; // f(z)=z^3+c
11         m2 = QuatLength2(z);
12         if (m2 > 256.0) break;
13         n += 1.0;
14     }
15     float d = 0.25 * log(m2) * sqrt(m2 / dz2);
16     return float2(d, n);
17 }
```

`CalcNormal()` (Listing 33-5) implements Equation 33.7 to approximate the gradient ∇d of the distance function d (Equation 33.3); $\nabla d(P)$ aligns with the normal at P of the isosurface with a regular value $d(P)$.

Listing 33-5. *Normal calculation.*

```
1 float3 CalcNormal(in float3 p)
2 {
3     float h = 0.5773f * kPrecis; // kPrecis = 0.00025f
4     const float2 v = float2(1.0f, -1.0f) * h;
5     return normalize(
6         v.xyy * Dist(p + v.xyy).x + v.yyx * Dist(p + v.yyx).x +
7         v.yxy * Dist(p + v.yxy).x + v.xxx * Dist(p + v.xxx).x );
8 }
```

Listing 33-6 shows the closest-hit shader for the Julia set. It uses a traditional approach, defining an albedo for the Phong model, combining it with a reflection color, and accumulating it with previous reflections.

Listing 33-6. *Closest-hit shader.*

```
1  [shader("closesthit")]
2  void ClosestHitJulia(inout RayPayload rayPayload,
3      in ProceduralPrimitiveAttributes attr)
4  {
5      // Albedo
6      float3 hitPosition = HitWorldPosition();
7      float3 pos = ObjectRayPosition();
8      float3 dir = WorldRayDirection();
9      float4 albedo = float4(3.5 * ColorSurface(pos, attr.color.xy), 1);
10     if (rayPayload.recursionDepth == MAX_RAY_RECURSION_DEPTH - 1)
11        albedo += 1.65 * step(0.0, abs(pos.y));
12
13     // Reflection
14     float4 reflectedColor = float4(0, 0, 0, 0);
15     float reflecCoef = 0.1;
16     Ray reflectionRay = { hitPosition,
17         reflect(WorldRayDirection(), attr.normal)};
18     float4 reflectionColor = TraceRadianceRay(reflectionRay,
19         rayPayload.recursionDepth);
20     float3 fresnelR = FresnelReflectanceSchlick(WorldRayDirection(),
21         attr.normal, albedo.xyz);
22     reflectedColor= reflecCoef * float4(fresnelR,1) * reflectionColor;
23
24     // Final color
25     float4 phongColor = CalculatePhongLighting(albedo, attr.normal);
26     float4 color = phongColor + reflectedColor;
27     color += rayPayload.color;
28     rayPayload.color = color;
29 }
```

33.3.2 MANDELBULB

In Section 33.1.2, we defined the Mandelbulb fractal of the polynomial function $f(P) = P^k + C$. Remember that a point $C \in \mathbb{R}^3$ belongs to the Mandelbulb if the recurrence $f^n(0) = (f^{n-1}(0))^k + C$ does not diverge. The most popular choice in the fractal community for rendering is $k = 8$.

The implementations of the Mandelbulb and Julia set (Section 33.3.1) are very similar. The major difference is in the function `Dist()`, which computes the iterations using Equation 33.1 to estimate the distance function. Listing 33-7 shows the code.

Given the point C, the loop (lines 7–17) iterates $(0, 0, 0)$ using the formula $f(P) = P^8 + C$. The eighth power of P is computed in lines 10–13 using the

Listing 33-7. *Mandelbulb distance.*

```
1  float Dist( in float3 c, out float4 resColor )
2  {
3      float3 w = c;
4      float m = dot(w,w);
5      float4 colorParams = float4(abs(w),m);
6      float dz = 1.;
7      for( int i=0; i<4; i++ )
8      {
9          dz = 8.0*pow(sqrt(m),7.0)*dz + 1.0;
10         float r = length(w);
11         float b = 8.0*acos( w.y/r);
12         float a = 8.0*atan2( w.x, w.z );
13         w = pow(r,8) * float3(sin(b)*sin(a),cos(b),sin(b)*cos(a)) + c;
14         colorParams = min( colorParams, float4(abs(w),m) );
15         m = dot(w,w);
16         if(m > 256.0) break;
17     }
18     resColor = float4(m,colorParams.yzw);
19     return 0.25*log(m)*sqrt(m)/dz;
20 }
```

formula given in Equation 33.1. The iterations are accumulated in line 13. If the new point diverges, we stop the loop in line 16. The distance is computed in line 19 using the formula in Equation 33.3 because its derivation can be applied to the Mandelbulb. The parameters used to define the colors for different parts of the fractal in the closest-hit shader, later on, are also computed in lines 5 and 14.

Listing 33-5 can also be used to approximate the normal vectors of the Mandelbulb surface. We just have to change the distance function by the one in Listing 33-7.

The Mandelbulb's closest-hit shader uses the color parameters computed by the distance function and the normal. Because the code derives from several empiric tweaks of parameter weights, we avoid listing it here. Details can be found in the full shader implementation (see the link in Section 33.4).

We can create procedural scenes containing both the Julia set and the Mandelbulb. To render such a scene, we consider that each fractal is inside an AABB. Figure 33-8 presents an example containing a Julia set, a Mandelbulb, a parallelepiped, and two CSG Pac-men.

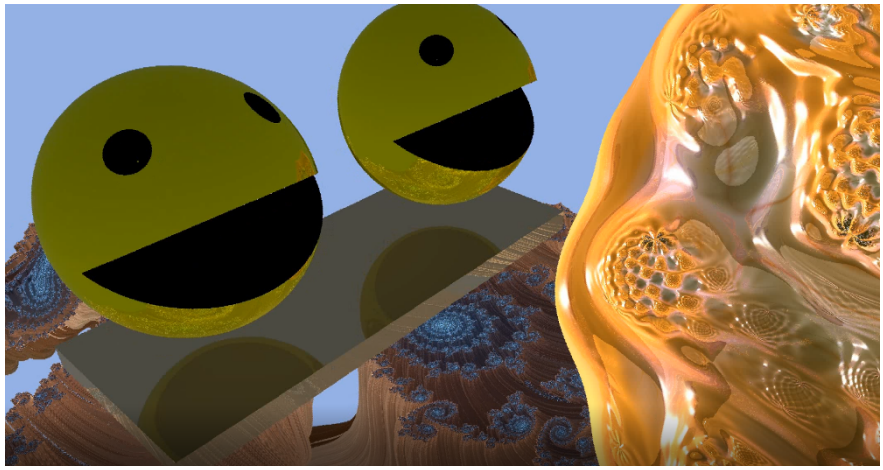


Figure 33-8. *Example scene: a triangle parallelepiped mesh and several procedural objects (two Pac-men, a 3D slice of a quaternion Julia set, and a Mandelbulb).*

33.4 CONCLUSION

This chapter described how to use DXR shaders to implement real-time renderings of complex fractals. Using custom intersection shaders enables scenes containing both fractals and triangle-based objects, which can be path-traced automatically using hardware acceleration. This work also compiled the associated mathematical tools in a brief but intuitive way.

Other complex fractals can be visualized in 3D with approaches similar to the ones in this chapter. They include the Juliabulb and the 3D slices of the quaternion Mandelbrot set. Additionally, exploring ways to visualize octonion fractals [9] in 3D could be an interesting path. Finally, developing level of detail approaches for those fractals could be interesting performance-wise [4].

The full code for shaders, including ray generation and miss, is available at github.com/dsilvavinicius/realtime_rendering_of_complex_fractals. For host code setup, please refer to Proceduray [3].

ACKNOWLEDGMENTS

Thanks to Iñigo Quilez for the shader used to generate Figures 33-1 and 33-5.

REFERENCES

[1] Artz, R. E. Scalar algebras and quaternions: An approach based on the algebra and topology of finite-dimensional real linear spaces, Part 1. https://www.math.cmu.edu/~wn0g/qu1.pdf, April 23, 2009.

[2] Bálint, C. and Valasek, G. Accelerating sphere tracing. In *Eurographics 2018—Short Papers*, pages 29–32, 2018. DOI: 10.2312/egs.20181037.

[3] Da Silva, V., Novello, T., Lopes, H., and Velho, L. Proceduray—A light-weight engine for procedural primitive ray tracing. Preprint, https://arxiv.org/abs/2012.10357, 2020. Engine available at https://github.com/dsilvavinicius/Proceduray.

[4] De Figueiredo, L. H., Nehab, D., Stolfi, J., and de Oliveira, J. B. S. Rigorous bounds for polynomial Julia sets. *Journal of Computational Dynamics*, 3(2):113, 2016.

[5] Douady, A. Julia sets and the Mandelbrot set. In H.-O. Peitgen and P. H. Richter, editors, *The Beauty of Fractals*, pages 161–174. Springer, 1986.

[6] Ebb, M., Sutherland, R., Heckenberg, D., and Green, M. Building detailed fractal sets for "Guardians of the Galaxy Vol. 2". In *ACM SIGGRAPH 2017 Talks*, 12:1–12:2. 2017. DOI: 10.1145/3084363.3085060.

[7] Failes, I. Mandelbulbs, mutations and motion capture: The visual effects of Annihilation. https://vfxblog.com/2018/03/12/mandelbulbs-mutations-and-motion-capture-the-visual-effects-of-annihilation/, 2018.

[8] Galin, E., Guérin, E., Paris, A., and Peytavie, A. Segment tracing using local Lipschitz bounds. *Computer Graphics Forum*, 39(2):545–554, 2020. DOI: 10.1111/cgf.13951.

[9] Griffin, C. and Joshi, G. C. Octonionic Julia sets. *Chaos, Solitons & Fractals*, 2(1):11–24, 1992. DOI: 10.1016/0960-0779(92)90044-N.

[10] Hart, J. C. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996.

[11] Hart, J. C., Sandin, D. J., and Kauffman, L. H. Ray tracing deterministic 3-D fractals. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, pages 289–296, 1989.

[12] Hutchins, D., Riley, O., Erickson, J., Stomakhin, A., Habel, R., and Kaschalk, M. Big Hero 6: Into the portal. In *ACM SIGGRAPH 2015 Talks*, 52:1. 2015. DOI: 10.1145/2775280.2792521.

[13] Kim, A., Ferreira, D. P., and Bevins, S. Capturing the infinite universe in "Lucy": Fractal rendering in film production. In *ACM SIGGRAPH 2014 Talks*, 1:1, 2014. DOI: 10.1145/2614106.2614166.

[14] Museth, K. VDB: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics*, 32(3):27:1–27:22, 2013. DOI: 10.1145/2487228.2487235.

[15] Norton, A. Julia sets in the quaternions. *Computers & Graphics*, 13(2):267–278, 1989. DOI: 10.1016/0097-8493(89)90071-X.

[16] Nylander, P. Hypercomplex fractals. http://www.bugman123.com/Hypercomplex/index.html, 2017.

[17] Quilez, I. 3D Julia sets. https://www.iquilezles.org/www/articles/juliasets3d/juliasets3d.htm, 2020.

[18] Quilez, I. Complex dynamics. https://www.shadertoy.com/view/MdX3zN, June 18, 2013.

[19] Quilez, I. Distance to fractals. https://www.iquilezles.org/www/articles/distancefractals/distancefractals.htm, 2004.

[20] Seymour, M. Doctor Strange's Magical Mystery Tour in time. https://www.fxguide.com/fxfeatured/dr-stranges-magical-mystery-tour-in-time/, November 14, 2016.

[21] White, D. The unravelling of the real 3D Mandelbulb. https://www.skytopia.com/project/fractal/mandelbulb.html, August 11, 2009.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 34

IMPROVING NUMERICAL PRECISION IN INTERSECTION PROGRAMS

Ingo Wald
NVIDIA

ABSTRACT

This chapter describes a useful and widely applicable technique for improving numerical stability in ray/primitive intersection tests. The described technique is all but novel, but is not well described in any literature, and thus not known to many practitioners—something this chapter aims to remedy.

34.1 THE PROBLEM

Scientific codes generally rely on double-precision floating-point calculations, but for performance reasons in graphics we typically use single-precision floats. Single precision is good enough for most things including primitive coordinates, ray origins and directions, barycentrics, colors, etc. However, in ray tracing one area where single precision frequently leads to issues is user intersection programs, in particular in situations where the primitive to be intersected is, relative to its size, far away from the ray origin. Such situations often lead to numerical precision errors in computing the intersection outcome and/or hit distance, often leading to artifacts that appear like concentric rings around the image center (such concentric rings are usually a dead giveaway for this kind of issue).

This problem of numerical stability in intersection tests has been previous addressed by two *Ray Tracing Gems* chapters—one by Wächter and Binder [2], and one by Haines et al. [1]. These two chapters cover triangles and spheres, respectively, and work well for those—but do not necessarily generalize. This chapter describes a completely orthogonal method to address this problem; it is easy to apply and is applicable to practically any intersection program. Though not at all novel (this author is aware of multiple prior uses by different practitioners), this method seems to not be as widely known as this generality would suggest it should be.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_34

34.2 THE METHOD

To fully understand *why* (and how) the eventual method will work, it helps to first properly understand the root cause of why such intersection tests for faraway primitives are problematic in the first place. To do this, we need to briefly look at floating-point numbers: Each floating point is represented by a mantissa and an exponent, and when two floating-point values get added (or subtracted, compared, etc.), the floating-point unit first needs to bring both terms to the same exponent. This is done by shifting the mantissa bits of the smaller value until the exponent matches that of the larger value—but shifting the mantissa means that some of the smaller value's mantissa lower bits get lost: a process called *vanishing* or *extinction* of the lower-value bits. How many bits get lost depends on how big the difference of the two exponents is: the greater that difference, the more bits of the lower value get lost (in extreme cases, the lower value may get lost completely).

In ray/primitive intersection computations this typically happens as soon as the primitive is far away from the ray origin: no matter the type of primitive, some of the terms computed in the intersection program will almost surely only involve values of the primitive itself (e.g., the radius of a sphere, the distance between two vertices or control points, etc.), while others will involve both ray origin and some primitive value (e.g, the ray origin's distance to the sphere center, or to a vertex, etc.). If the object is far away relative to its size, these two types of terms will then have very different magnitudes, and any operation between them will lead to extinction. Generally speaking, the farther away and/or the smaller the object, the worse the problem; and if the computations use the square (or any other power) of such values, it only gets worse.

Once we understand this root cause, we an simply address it by temporarily moving the ray origin closer to the object: for any ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, any other origin $\mathbf{o}'$ on that same ray should actually yield the same intersection, except for an obviously different hit distance; yet, if that temporarily shifted origin is much closer to the object, the origin-primitive terms should get significantly smaller, reducing the previously described extinction effects.

This trick is trivial to apply: First, we compute *any* approximate distance t_{approx} to the primitive; this value does not have to be exact, just something that produces an origin anywhere reasonably close to the primitive. Second, we compute a new, temporary ray $\mathbf{r}'(t) = \mathbf{o}' + t\mathbf{d}$ with same ray direction $\mathbf{d}$ but

with origin $\mathbf{o}' = \mathbf{o} + t_{\text{approx}}\mathbf{d}$ and with similarly adjusted ray interval $[t'_{\text{min}}, t'_{\text{max}}] = [t_{\text{min}} - t_{\text{approx}}, t_{\text{max}} - t_{\text{approx}}]$. We then perform the intersection computations with this "shifted" ray—but because the origin-primitive terms should now be on the same scale as primitive-only terms, extinction will be significantly less. Finally, if r' did find a valid intersection at t'_{hit}, we know that the original ray r should have hit at $t_{\text{hit}} = t'_{\text{hit}} + t_{\text{approx}}$, so all we have to do is adjust the returned hit distance and report this (barycentrics, normal, etc. are not affected by this method).

34.2.1 IMPLEMENTATION NOTES

Implementing this technique is trivial, too: given an existing intersection program `intersect_naive(ray,prim)`, all we need to do is supplement this program with a second intersection program that temporarily shifts the ray origin, then calls the original one, and re-adjusts the hit distance:

```
bool intersect_naive(Prim prim, Ray ray, Hit hit)
{ ... /* arbitrary intersection program */ ... }

bool intersect_improved(Sphere sphere, Ray ray, Hit hit)
{
  // Compute _any_ approximate distance.
  float tApprox = /* any approximate distance to prim */...;
  // ``Temporarily'' shift the ray origin closer to the object.
  float3 shiftedOrigin = ray.direction + tApprox*ray.direction;
  Ray shiftedRay(shiftedOrigin, ray.direction,
                 ray.tMin - tApprox, ray.tMax - tApprox);
  // Call naive test with the shifted ray.
  if (!sphereTest_naive(sphere,shiftedRay,hit))
     return false;
  // Shift the computed hit point back.
  // In OptiX: optixReportIntersection(hit.t+tApprox)
  hit.t += tApprox;
  return true;
}
```

To demonstrate the effectiveness of this trick, Figure 34-1 (taken from [3]) shows an example of it being applied to ray/curve intersections. Without shifting the origin, the accuracy of the curve intersection diminishes very quickly, because terms with the curve radius vanish relative to the origin-curve distance terms—leading to some curves being missed while others report false positives, with very inaccurate intersection distances that lead to wrong secondary rays.[1]

[1] These pictures also exhibit the telltale circular artifacts around the image center that we mentioned in Section 34.1.



Figure 34-1. *Impact of the technique described in this chapter on a ray/curve intersector in [3]. Left: intersection test with the original ray, leading to very inaccurate intersection tests. Right: with this chapter's technique and otherwise unchanged intersection test.*

34.2.2 WHICH DISTANCE TO CHOOSE?

So far, we have not said much about what distance t_{approx} one should use. In fact, it does not actually matter much, as long as it reduces the magnitude of the origin-to-object distance term(s); for example, for a sphere, cylinder, torus, etc., an obvious candidate is the distance to the object's center; for any object with control points, the distance to any control point should do; for code that involves any sort of bounding box computation and test, the distance to that bounding box should be a good candidate; etc.

Even more generally, if one does have access to the bounding volume hierarchy traverser's distance to the leaf node that contained the primitive (or if one used any sort of early-out bounding box test), then the distance to this bounding volume is an excellent choice. In OptiX or DirectX Raytracing this distance to the bounding box is not yet exposed to the intersection program; but if it ever will be, it would be a good candidate.

34.2.3 LIMITATIONS AND PITFALLS

Though the main strength of this technique is its wide applicability, it is not a panacea: it cannot solve all numerical precision issues and should not be an excuse to not (also) use better numerical techniques if and where available (see, e.g., the aforementioned chapters by Haines et al. [1] and Wächter and Binder [2]).

One possible pitfall is that in cases where primitives are far away from the *ray* origin, these primitives will almost surely also be far away from the origin of the coordinate system—in which case extinction can still happen between terms that combine large absolute primitive coordinates with much smaller primitive values like edge lengths, radii, etc. Interestingly, the trick just described often still works, as long as the intersection test first shifts the primitive into a coordinate system centered around the ray origin. Many primitive tests already do that for performance reasons (a ray origin at (0, 0, 0) simplifies many terms, leading to cheaper intersection tests), but those that do not may need to address those issues separately.

34.2.4 SUMMARY

We have discussed a simple trick for reducing floating-point extinction effects in arbitrary ray intersection tests. The trick is trivially easy to apply, virtually free in terms of computations, and has, in this author's own experience, proven itself to work well for spheres, cylinders, tubes, curves, and others. Though it cannot solve all problems, it is a trick that this author believes every ray tracing practitioner should at least be aware of.

ACKNOWLEDGMENTS

This author first learned this technique from Johannes Günther, who applied it to improve various primitive intersection tests in the OSPRay system. The images in Figure 34-1 were created with help from Nate Morrical and Stefan Zellmann and using data provided by the Blender Foundation.

REFERENCES

[1] Haines, E., Günther, J., and Akenine-Möller, T. Precision improvements for ray/sphere intersection. In E. Haines and T. A. Möller, editors, *Ray Tracing Gems*, chapter 7, pages 86–94. Apress, 2019.

[2] Wächter, C. and Binder, N. A fast and robust method for avoiding self-intersection. In E. Haines and T. A. Möller, editors, *Ray Tracing Gems*, chapter 6, pages 77–85. Apress, 2019.

[3] Wald, I., Morrical, N., Zellmann, S., Ma, L., Usher, W., Huang, T., and Pascucci, V. Using hardware ray transforms to accelerate ray/primitive intersections for long, thin primitive types. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2):17:1–17:16, 2020. DOI: 10.1145/3406179.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 35

RAY TRACING OF BLOBBIES

Manuele Sabbadin and Marc Droske
Weta Digital

ABSTRACT

Particles are widely used in movie production rendering for various different effects. Blobbies (aka metaballs) are a very useful primitive to bridge the intermediate regime between the bulk of a fluid and fast-moving spray particles as well as providing geometric variation to droplets. The use of anisotropy allows one to represent thin line structures better than classic isotropic shapes. Tessellation of such fine geometric structures is prone to geometric artifacts, especially under strong motion blur, which may heavily distort the surface during the shutter or because topology changes can't be represented well. Intersecting rays with the isosurface analytically has robustness and precision advantages. Operating on the original representation provides highly accurate spatial and temporal derivatives that are useful for filtering specular highlights. In this chapter we describe some algorithmic tools to robustly and efficiently intersect blobby surfaces supporting anisotropy and higher-order motion blur.

35.1 MOTIVATION

High-quality rendering of special effects elements (see Figure 35-1) requires robust and accurate representation of geometric details at various different scales. Elements such as fine spray can be represented by volumes and particles, whereas for the bulk of a fluid, morphological surfacing techniques can successfully be applied [8]. High-frequency details of implicit surfaces, especially under motion, pose challenges for tessellation-based techniques due to distortions and topology changes, which might require a large amount of motion steps to mask and special care to avoid artifacts due to potential self-intersections. Furthermore, high curvature and temporal variation of the normal cause specular highlights to be challenging to resolve. Here, spatial [6, 11] and temporal [10] antialiasing techniques yield very good results. These rely on surface derivatives to estimate the normal variation to translate into Beckmann roughness and to compute ray differentials. In

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_35



Figure 35-1. *Blobbies are largely used in visual effects to represent splashes of water. Anisotropy of the particles allows one to preserve the correct shape of the thin walls and lines of water. In this image 1,221,370 blobbies represent an exploding bowl of water.*

particular, we rely on computing up to second higher-order and mixed derivatives to be computed reliably, which can be a challenge on its own.

Blobby surfaces, as first introduced by Blinn [1], offer an analytic definition of an isosurface based on the combination of kernel functions around given particles (see Figure 35-2). The resulting surfaces are smooth and are nicely and compactly represented by points with some parameters. Motion can be expressed in a very natural way in a Lagrangian formulation. However, in their basic form the resulting surfaces are more suitable for molecular visualization than for fluids. Their wobbly appearance makes it difficult to represent thin structures. The definition of blobbies extends, however, easily to anisotropic surfacing [12], which overcomes these issues and makes them a compelling modeling representation for various forms of splashes and fluid droplets.

Because the surface is implicitly defined by particles that interact with each other, finding the intersections both robustly and efficiently requires some extra care. We describe some ingredients for using ray traced blobbies in practice:

- Revisit anisotropic blobby particles.
- Bounding volume hierarchy (BVH) traversal tailored to interval refinement methods [9].
- Computing tight bounds of individual blobby functions.

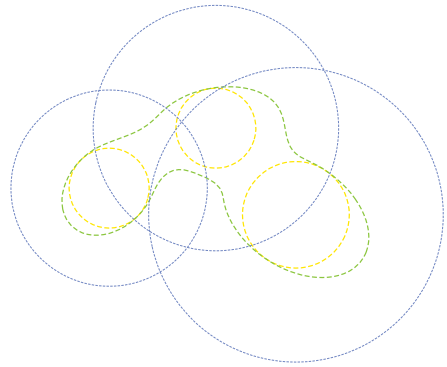


Figure 35-2. *Blobby field from three anisotropic particles with corresponding isosurface (green). With each particle we associate an inner sphere (orange) and a bounding sphere (blue), the smallest sphere that contains the support region.*

35.2 ANISOTROPIC BLOBBIES

A blobby particle B_i is represented by an implicit field $\psi_i : [t_0, t_1] \times \mathbb{R}^d \to \mathbb{R}$ defined on time in $[t_0, t_1]$ and space. A set of blobbies defines an implicit field $\phi(t, \mathbf{x})$ in the following way:

$$\phi(t, \mathbf{x}) = \sum_i \psi_i(t, \mathbf{x}) - T, \tag{35.1}$$

where T is a threshold parameter that influences the blending of the different blobbies (see Figure 35-2). To visualize $\phi(t, \mathbf{x})$, we are interested in the isosurface defined by $\mathcal{M}(t) = \{\phi(t, \cdot) = 0\}$.

We focus on the classic blobby variant

$$\psi_i(t, \mathbf{x}) = \begin{cases} \left(1 - R_i^{-2} \|\mathbf{x} - \mathbf{x}_i(t)\|^2\right)^3 & \|\mathbf{x} - \mathbf{x}_i\| < R_i, \\ 0 & \text{otherwise}, \end{cases} \tag{35.2}$$

where $\mathbf{x}_i(t)$ defines the center of the particle at time t and R_i is the radius of the influence region. We denote with R_i the *bounding radius* of the blobby B_i.

This can easily be generalized to anisotropic particles by writing it in the form

$$\psi_i(t, \mathbf{x}) = k\left(g\left(\mathbf{x} - \mathbf{x}_i(t), \mathbf{x} - \mathbf{x}_i(t)\right)\right), \quad \text{where} \quad k(y) = (1 - y)^3 \tag{35.3}$$

and g is a scalar product $g_A(\mathbf{u}, \mathbf{v}) = \langle A\mathbf{u}, A\mathbf{v} \rangle$ that encodes the anisotropy and size. In particular, for a unit basis $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2$ and radii R_0, R_1, R_2, we can set

$A_i = \text{diag}(\frac{1}{R_0}, \frac{1}{R_1}, \frac{1}{R_2}) \cdot [\mathbf{b}_0\ \mathbf{b}_1\ \mathbf{b}_2]^T$. The values R_i can be easily chosen depending on T such that the isosurface of an isolated blobby describes an ellipsoid with the prescribed lengths r_1, r_2, r_3 of the axes. In the case of anisotropic particles, the *bounding radius* is defined as the maximum of $\{R_0, R_1, R_2\}$. We also define r_i, the *inner radius* of the blobby B_i, as the radius of the largest sphere that is contained in the isosurface, if B_i is not influenced by any other blobby. It is equal to the minimum value of $\{r_1, r_2, r_3\}$.

Expressions like the shape operator $\mathcal{S}$, the temporal derivative of the normal, or the derivative of the intersection distance (see [10]) are easily obtained through basic derivatives of the level set function such as $\partial_t \phi$, $\nabla_x \phi$, $\text{Hess}\, \phi$, and $\partial_t \nabla_x \phi$. In particular, the shape operator corresponds to the matrix representation of the *Weingarten map*:

$$\mathcal{S} = \frac{1}{\|\nabla_x \phi\|} P[\nabla_x \phi]\, \text{Hess}\phi\, P[\nabla_x \phi], \quad \text{where} \quad P[v] = (\text{id} - v \otimes v), \tag{35.4}$$

which is useful to compute normal derivative in direction v as $D_v N = \mathcal{S}\, v$.

Setting $g_i(t, x) = \langle A_i(x - x_i(t)), A_i(x - x_i(t)) \rangle$, we have, for example,

$$\partial_t \nabla_x \psi_i(t, x) = k''(g_i(t, x))\, \partial_t g_i(t, x)\, \nabla_x g_i(t, x) + k'(g_i(t, x))\, \partial_t \nabla_x g_i(t, x), \tag{35.5}$$

where

$$\begin{aligned} \partial_t g_i(t, x) &= -2 \left\langle A(x - x_i(t)), A\, \partial x_i(t) \right\rangle, \\ \nabla_x g_i(t, x) &= 2A^T A(x - x_i(t)), \\ \partial_t \nabla_x g_i(t, x) &= -2A^T A\, \partial_t x_i(t). \end{aligned} \tag{35.6}$$

Motion blur is expressed simply as a parametric form of the center $x_i(t)$ depending on t. The equations can easily be extended to support a time-dependent metric $A(t)$ to represent, for example, oscillations and spin.

35.3 BVH AND HIGHER-ORDER MOTION BLUR

We use a classic BVH to store blobbies and to identify particles whose supports overlap with a ray segment. Each blobby is represented as a sphere inside the BVH (even for the anisotropic case, as we will discuss in Section 35.4). For a generic blobby B_i, we store its bounding radius R_i and the inner radius r_i. To tackle the motion of each blobby, we also store its velocity v_i and acceleration a_i at time t_0. This will allow us to represent higher-order motion blur, instead of just a linear motion. During the construction of a BVH, it is important to create bounding boxes as tight as possible to the actual

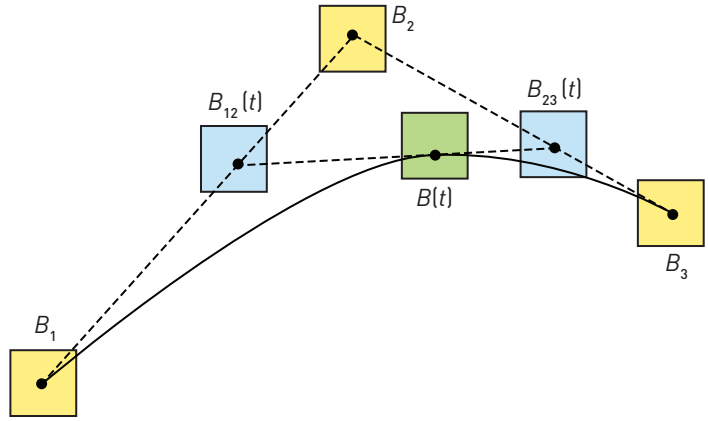


Figure 35-3. *Higher-order motion bounds: Given the bounds B_1, B_2, and B_3 that contain the control points of the input curve, $B_{12}(t) = (1 - t)B_1 + tB_2$ and $B_{23}(t) = (1 - t)B_2 + tB_3$ are bounds of the two control points of the first iteration of the de Castlejau interpolation. Eventually, $B(t) = (1 - t)B_{12}(t) + tB_{23}(t)$ contains the curve evaluated at time t.*

geometry. When objects are moving, this task becomes trickier. The simplest solution, using a bounding box that contains the entire trajectory of the object, will become inefficient under fast motion as bounds for a specific ray time become loose. A better option is to exploit velocity and acceleration to interpolate bounds in time on leaf nodes and to propagate this motion higher up the tree.

Linear motion blur can easily be handled by linearly interpolating the bounding boxes during traversal according to the ray time [3], which yields much tighter bounds than growing the bounding boxes to contain the entire motion path. Because de Casteljau's algorithm is an iterated version of linear interpolation, this can easily be extended to higher-order Bézier interpolation: the control points can be used to define control bounding boxes that are Bézier-interpolated during the traversal (see Figure 35-3).

Therefore, we only need to convert the incoming parabola given by velocity v and acceleration a defined on [0, 1] as

$$\mathbf{x}_i(t) = \mathbf{p}_i(0) + t\mathbf{v}_i + \frac{1}{2}t^2\mathbf{a}_i \tag{35.7}$$

into Bézier form:

$$\mathbf{B}_i(t) = (1 - t)^2\mathbf{P}_{i,0} + 2(1 - t)t\mathbf{P}_{i,1} + t^2\mathbf{P}_{i,2}. \tag{35.8}$$

The change of basis is given by $\mathbf{P}_{i,0} = \mathbf{x}_i(0) = \mathbf{p}_i$, $\mathbf{P}_{i,1} = \frac{1}{2}\mathbf{v}_i + \mathbf{P}_{i,0}$ and $\mathbf{P}_{i,2} = \frac{1}{2}\mathbf{a}_i + \mathbf{v}_i + \mathbf{p}_i = \mathbf{x}_i(1)$. From these the bounds of the control points for each

leaf node are obtained, which defines three Bézier control bounding boxes. Again analogously to the linear interpolation case, the control bounding boxes are aggregated up toward the root.

35.4 INTERSECTION METHODS

We define $r = (\mathbf{o}, \mathbf{d})$ as the ray with origin $\mathbf{o}$, direction $\mathbf{d}$, and limits $[s_{\text{min}}, s_{\text{max}}]$, parameterized as $r(s) = \mathbf{o} + s\mathbf{d}$.[1] We want to determine an intersection distance $s_{\text{int}} \geq 0$ that is the minimum $s \in [s_{\text{min}}, s_{\text{max}}]$ such that $\phi(t, r(s_{\text{int}})) = 0$. We indicate the hit point with $\mathbf{h}$. For an interval I (along the ray) in which $\phi(t, r(s))$ assumes both positive and negative values and is monotone, we can apply any root-finding method to find $\mathbf{h}$ within that interval. We can consider the bounding box enclosing the entire set of blobbies and intersect it with the ray r, finding a first guess for such an interval. Then, bisection can be applied iteratively until it satisfies the above-mentioned conditions. This approach works in theory, but it is not feasible in practice, as the amount of blobbies that define the implicit field ϕ can be very large. This means that every time we want to evaluate $\phi(t, \mathbf{x})$, we need to sum up the contribution from all the blobbies. Any root-finding algorithm would require $\phi(t, \mathbf{x})$ to be computed more than once, at different points along the ray. Given the finite support of the kernel function, only a few of them will contribute to the value $\phi(t, \mathbf{x})$. Naturally, we can restrict the number of blobbies we use to those whose bounding boxes intersect r. Even so, we are potentially considering a large amount of blobbies that are too far away from $\mathbf{h}$ to contribute to its computation. Our aim is to find the set of blobbies $\mathcal{A}$ required by the root-finding algorithm (i.e., it contains all the blobbies B_i for which $\psi_i(t, \mathbf{h}) \neq 0$) while discarding as many as possible.

The main steps of our algorithm are the following:

1. Determine $\mathcal{A}$ and $I_0 \subseteq [s_{\text{min}}, s_{\text{max}}]$, which is our first guess for the interval I.
2. Refine the interval I_0 iteratively, proceeding in front-to-back order, until we obtain an interval I_n that contains exactly one root.
3. Find the root inside the interval I_n.

We will focus on the first two steps, as the third consists of using a standard iterative root-finding algorithm.

[1] We use s to parameterize the ray to avoid conflict with the time denoted by t.

35.4.1 DETERMINE THE ACTIVE BLOBBIES

As we have previously seen, given a blobby B_i, we can define two different radii on it: The first one is the bounding radius R_i, which refers to the region of influence of B_i (i.e., $\psi_i(\mathbf{x}) = 0$ when $\|\mathbf{x} - \mathbf{x}_i\| > R_i$). The second radius is the inner radius r_i, which represents the radius of the largest sphere that is contained in the isosurface of B_i if not influenced by any other blobby. We associate a sphere to each of them: respectively, the bounding sphere $\mathcal{S}_{\text{bound},i}$ and the inner sphere $\mathcal{S}_{\text{inner},i}$. Each time we intersect the ray r with B_i, we obtain four values: $S_i[\text{min}]$, $S_i[\text{max}]$, $s_i[\text{min}]$, and $s_i[\text{max}]$. The first two quantities are the values of s that lead to the two intersections on the bounding sphere $\mathcal{S}_{\text{bound},i}$, while the latter two refer to the intersections on the inner sphere $\mathcal{S}_{\text{inner},i}$. It is worth noting that we check for intersections against spheres even for the anisotropic case because the intersection test is computationally faster compared to a ray-ellipse test. As we need to be conservative in our criteria to discard a node, we set R_i to be the major radius of the bounding ellipse and r_i to be the minor radius of the inner ellipse.

To determine which blobbies we should consider and determine the interval I_0, we have to distinguish two different cases: if the ray is pointing inside (such as interior reflection or transmitting inward) or outside the surface. For both cases, we avoid adding a blobby B_i to the set $\mathcal{A}$ if the ray r does not intersect $\mathcal{S}_{\text{bound},i}$. Moreover, we traverse our BVH in a way that prioritizes the nodes closer to the ray origin, as shown in the following listing:

```
void VisitChildren(Stack& S, Ray r, Node n) {
    float t0 = r.GetNearHitpointBounding(n.Child[0]);
    float t1 = r.GetNearHitpointBounding(n.Child[1]);

    int furthest = 0;
    if (t1 > t0) furthest = 1;

    if (r.IntersectBounding(n.Child[0]) && r.IntersectBounding(n.Child[1]))
         {
        S.Push(n.Child[furthest]);
        S.Push(n.Child[1 - furthest]);
        return;
    }
    if (r.IntersectBounding(n.Child[0]))
        S.Push(n.Child[0]);
    if (r.IntersectBounding(n.Child[1]))
        S.Push(n.Child[1]);
}
```

TRACING TOWARD FRONTFACE

When we hit the isosurface from the outside, we can state that if we hit the inner sphere $\mathcal{S}_{\text{inner},i}$ of the blobby B_i, we don't need to add to $\mathcal{A}$ any node B_j

that is farther from the origin of r and whose bounding sphere $\mathcal{S}_{\text{bound},j}$ does not intersect $\mathcal{S}_{\text{inner},i}$. This is simply motivated by the fact that the blobby B_j cannot influence in any way the implicit field around the hit point (and cannot cover it because it is farther). This can be achieved by a simple check: we can discard B_j if $S_j[\text{min}] > s_i[\text{min}]$. The following listing shows a simple implementation of the method to find the closest hit if the ray origin lies outside the isosurface; we use a stack to keep track of the traversal's state:

```
// The BVH B stores all the blobbies.
void IntersectFromOutside(Ray r, Set A) {
    Stack S;
    S.Push(B.root);
    float tmax = FLT_MAX;

    while (!S.IsEmpty()) {
        Node n = S.Pop();
        // If the bounding box is farther than the hit point, it cannot
             influence it.
        if (r.GetNearHitpointBounding(n) > tmax)
            continue;

        if (n.IsLeaf()) {
            if (r.IntersectInner(n)) {
                // t defines the intersection along the ray.
                float t = r.GetNearHitpointInner(n);
                tmax = min(tmax, t);
            }
            if (r.IntersectBounding(n))
                A.Insert(n);
        }
        else
            VisitChildren(S, r, n);
    }
}
```

TRACING TOWARD BACKFACE

When we hit the isosurface from the inside, we cannot use the same argument we used in the previous case. In this case, when we hit the inner sphere $\mathcal{S}_{\text{inner},i}$, there is no guarantee that nodes that satisfy the condition $S_j[\text{min}] > s_i[\text{min}]$ won't contribute to determining the hit point **h**. Let's imagine, for example, a chain of intersecting blobbies. If we start the ray r from one side of the chain, we have to traverse all the blobbies to detect the exit point (see Figure 35-9). In this scenario, we can use a weaker condition that allows us to discard part of the blobbies along the ray: Let B_i be the node in $\mathcal{A}$ with the largest $S_i[\text{max}]$. If all the nodes B_j that we still have to visit satisfy $S_j[\text{min}] > S_i[\text{max}]$, we can stop collecting nodes, because we must have exited in between. There is a crucial difference to the previous case, in which we could discard the single blobby and continue the visit the other nodes on

the stack. However, in this case we can only ascertain when the entire visit can be terminated, but it can't be determined whether a specific particle can be discarded. This is due to the fact that the entries in our stack are not ordered by *S*[min]. Hence, a node that we will visit later during the traversal could make the current node active, even if it is actually too far away to contribute to the hit (see Figure 35-9). We tested this approach on the data set in Figure 35-13 and compared to a version of the algorithm where we collect all the blobbies along the ray. The proposed technique gives a speedup of 24% for the render time.

One could argue that we should use a heap data structure instead of a stack to support the visit, to allow the algorithm to consider the closest entry at each iteration and to be able to stop the visit earlier. It is indeed an option that would allow for some optimizations, but the trade-off of the added cost of updating the heap might not be worth it.

The following listing shows a simple implementation of the method to find the closest hit if the ray origin is located inside the isosurface. Note that, to check if we can terminate the traversal, each entry of the stack now saves the minimum distance from the ray origin at the moment of its insertion.

```
1  // The BVH B stores all the blobbies.
2  void IntersectFromInside(Ray r, Set A) {
3      Stack S;
4      S.Push(B.root);
5      float tmax = FLT_MAX;
6      bool hasToInitTmax = true;
7
8      while (!S.IsEmpty()) {
9          if (S.Top().MinDistanceFromOrigin > tmax)
10             return;
11
12         Node n = S.Pop();
13
14         if (n.IsLeaf()) {
15             if (r.IntersectBounding(n)) {
16                 if (hasToInitTmax) {
17                     tmax = r.GetFarHitpointBounding(n);
18                     hasToInitTmax = false;
19                 }
20                 else
21                     tmax = max(r.GetFarHitpointBounding(n), tmax);
22                 A.Insert(n);
23             }
24         }
25         else
26             VisitChildren(S, r, n);
27     }
28 }
```

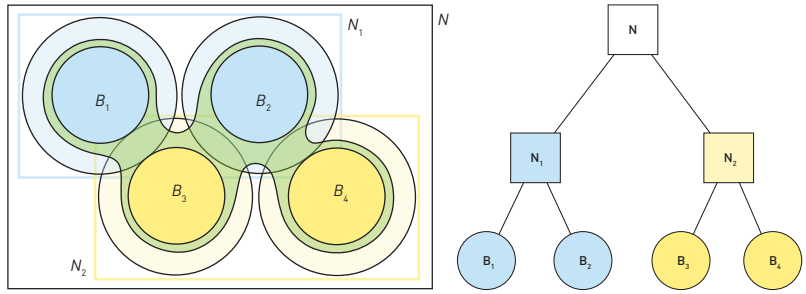


Figure 35-4. *Left: to better explain the different cases that we have to consider when creating the set $\mathcal{A}$, we will use a data set made by four blobbies, whose bounding spheres $\mathcal{S}_{\text{bound},i}$ intersect in pairs. In green we represent the isosurface. For each blobby, we used an opaque color to represent the inner sphere and a semitransparent one to represent the bounding sphere. Right: The BVH stores the nodes in a way that B_1 and B_2 share the same parent node N_1, while B_3 and B_4 share the same parent node N_2.*

We have to update the method `VisitChildren()` to push the value `MinDistanceFromOrigin` onto the stack, with each new entry. The value to push is the minimum between the old minimum distance (`MinDistanceFromOrigin` of the current top of the stack) and the distance to the node that we are going to push. For a generic node *n*, it works in the following way:

```
1 float minValue = S.Top().MinDistanceFromOrigin;
2 S.Push(n, min(minValue, r.GetNearHitpointBounding(n)));
```

EXAMPLES

In the following we provide a graphical representation to illustrate how the algorithm works. We will use the blobbies configuration in Figure 35-4 and change the ray origin and direction to present the most common situations. Figures 35-5 to 35-7 consider a ray hitting from outside the isosurface, whereas Figures 35-8 to 35-10 show a ray whose origin is inside it. In all the figures we will represent with a dotted contour the blobbies that are not part of the set $\mathcal{A}$ at the end of the traversal (for example, node B_4 in Figure 35-6).

35.4.2 INTERVAL REFINEMENT

From the set $\mathcal{A}$ we can easily find the interval $I_0 = [\min_0, \max_0]$: we simply have to intersect *r* against all the blobbies in the set and compute the boundaries of the interval in the following way:

$$\begin{aligned} \min_0 &= \textbf{Min}(S_i[\min]) \; \forall B_i \in \mathcal{A}, \\ \max_0 &= \textbf{Max}(S_i[\max]) \; \forall B_i \in \mathcal{A}. \end{aligned} \tag{35.9}$$

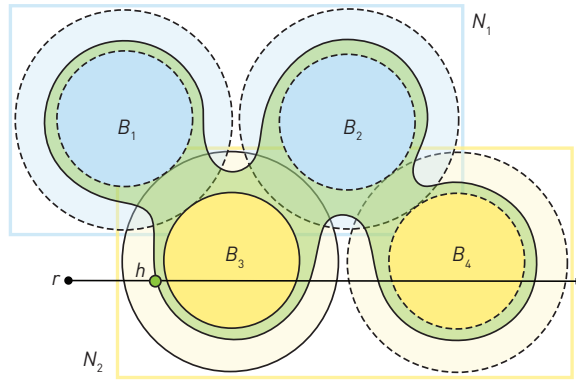


Figure 35-5. *The simplest case happens when the ray intersects an inner sphere $\mathcal{S}_{\text{inner},i}$ and no other sphere $\mathcal{S}_{\text{bound},j}$ intersects both the ray and $\mathcal{S}_{\text{inner},i}$. In the example, r does not intersect N_1, which is discarded immediately, with all its children. It first intersects $\mathcal{S}_{\text{inner},3}$ and adds the blobby to the set $\mathcal{A}$. When r intersects $\mathcal{S}_{\text{bound},4}$, the test $S_4[\min] > s_3[\min]$ succeeds, so we can discard it. The only active node is B_3.*

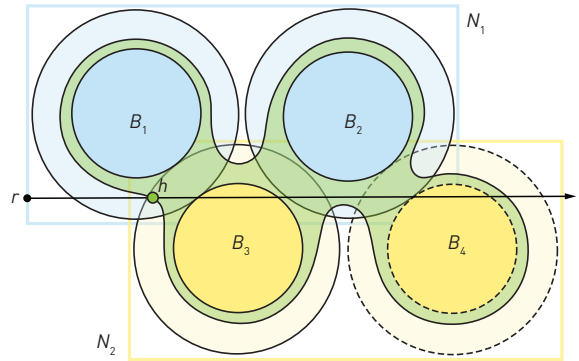


Figure 35-6. *When we intersect B_1, we don't intersect its inner sphere. In this case we cannot set an upper bound for the future intersections, as we don't know if B_1 will contribute to determining the hit point or not. Because we are visiting N_1, the next blobby we process is B_2, for which the same argument holds. When the algorithm processes N_2 and hits the inner sphere of B_3, it will set $s_3[\min]$ as an upper bound and discard B_4 at the next iteration, as $S_4[\min] > s_3[\min]$. The set of active nodes contains B_1, B_2, and B_3, even if B_2 won't contribute to computing the hit point.*

To guarantee that an interval I_i contains a single root, it is sufficient for ϕ to be monotone and the two extremes of the range B_I to have different sign. Therefore, to isolate the roots, one can use the well-known interval refinement approach as described, for example, in [7, 4]: we successively refine the ray segment until an interval is reached in which $s \mapsto \phi(t, r(s))$ is not

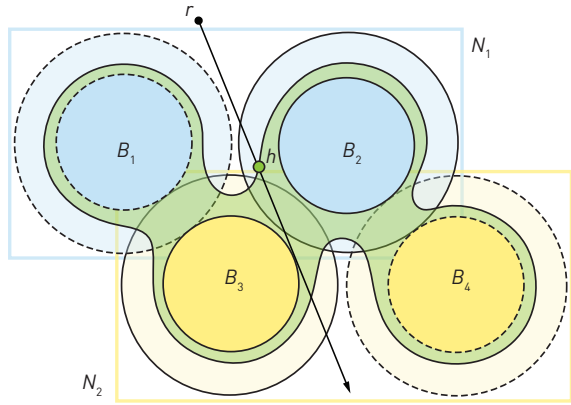


Figure 35-7. *The most unfortunate case, if we hit from outside, is when the ray r intersects many bounding spheres, but no inner spheres. In this example, we add to $\mathcal{A}$ both of the nodes B_2 and B_3 (which are required to compute the hit point). Because we are not able to set an upper bound, if r intersects any other blobby along is trajectory, it will be added to $\mathcal{A}$, no matter its distance from the origin. This cannot be avoided: there are cases where two bounding spheres intersects, but not enough to define the isosurface between them. In this case, the ray can pass in between the two blobbies and intersect something that is farther away.*

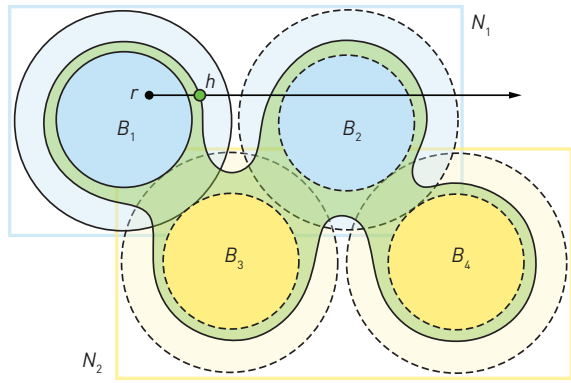


Figure 35-8. *When hitting the isosurface from inside, we can stop the process only if all the entries on the visiting stack are farther than the current upper bound. In this case, because r doesn't intersect N_2, this node doesn't appear on the stack. When we hit B_1, the only entry to compare with will be B_2 whose bounding sphere is not touching B_1. The traversal can stop immediately.*

bounded away from zero and is monotone (can't contain multiple roots), i.e., the derivative with respect to *s* is guaranteed not to be zero. The interval refinement can be done by bisection or an *Interval Newton method* [2], which uses the bounds of the derivatives in the refinement process.

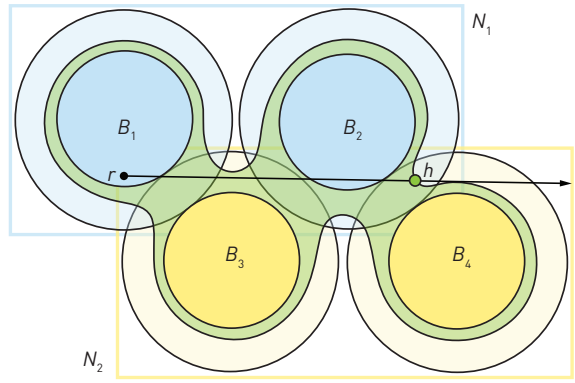


Figure 35-9. *In this example, we can see why the criteria used for an outside ray would not work for an inside ray. When we hit the inner sphere of B_1, we set S_1[max] as the upper bound. As B_2 does not intersect it, if hitting from outside, we would discard the node and keep going with the blobbies in N_2, ignoring the fact that the hit point is on its area of influence. This happens because we cannot know, beforehand, that there will be a node in N_2 acting as a bridge between B_1 and B_2 (B_3). Hence, the test for an inside ray checks all the entries on the stack.*

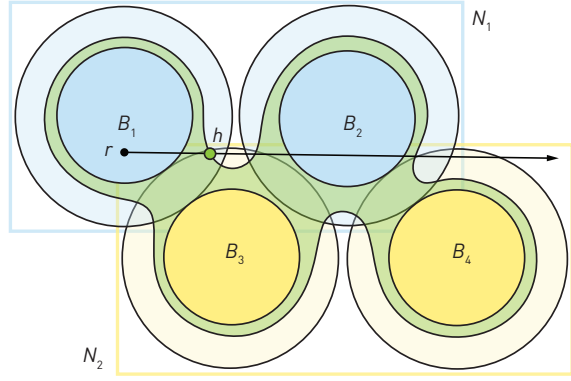


Figure 35-10. *An unfortunate case for the inside ray scenario is when we hit the isosurface soon, but we have a long chain of connected blobbies. The algorithm cannot know, a priori, that B_2 and B_4 are not needed to detect the hit point, so it will add them to the set $\mathcal{A}$, in the same fashion as Figure 35-9.*

Therefore, for a given ray $r(s) = \mathbf{o} + s\mathbf{d}$, the intersection distance interval relies on computing the bounds of $f(s) = \phi(t, r(s))$ and its derivative $f'(s)$ in an arbitrary subrange $[s_{\text{min}}, s_{\text{max}}]$. Of course, the efficiency of the refinement process depends on how tight the bounds are.

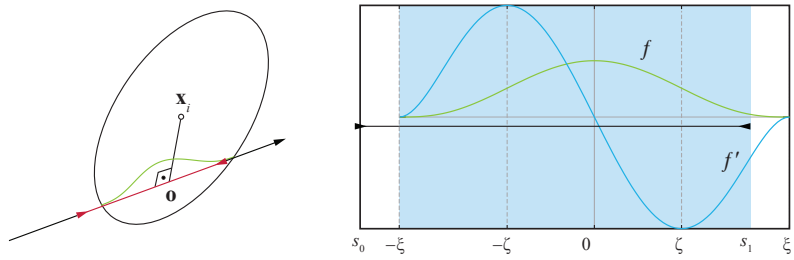


Figure 35-11. *Calculating bounds of ψ along the ray by identifying monotone intervals.*

Although interval arithmetic [4] could be applied for the computation of these bounds, this approach tends to produce too loose bounds. Instead, we proceed by computing tight bounds for the individual ψ_i (see Figure 35-11) and aggregate them through additive composition. Furthermore, we exploit the fact that $\psi_i \geq 0$ for early termination of the check for whether the bounds contain a root at all.

To compute tight bounds for ψ_i, one can simply exploit the well-known fact that the bounds of a monotone function are given by the values at the endpoints. We assume for simplicity that **o** is at the projection (with respect to g_A) of the center $x_i(t)$ to the ray

$$g_A(\mathbf{x}_i(t) - \mathbf{o}, \mathbf{d}) = 0, \tag{35.10}$$

by computing the projection and shifting $[s_{\text{min}}, s_{\text{max}}]$ accordingly.

Defining $\alpha = g_A(\mathbf{o} - \mathbf{x}_i, \mathbf{o} - \mathbf{x}_i)$ and $\beta = g_A(\mathbf{d}, \mathbf{d})$, we would then like to find the bounds of $f(s) = k(\alpha + s^2\beta) = (1 - \alpha - s^2\beta)^3$.

It can easily be seen that due to Equation 35.10 the support of f is $[-\xi, \xi]$ with $\xi = \sqrt{(1-\alpha)/\beta}$. It is monotonely increasing in $[-\xi, 0]$ and decreasing in $[0, \xi]$ and furthermore has inflection points at $-\zeta$, 0, and ζ with $\zeta = \sqrt{\frac{1}{5}}\xi$ (see Figure 35-11, right).

Therefore, computing the bounds on f in $I = [s_{\text{min}}, s_{\text{max}}]$ amounts to evaluating at the endpoints of the subintervals $[-\xi, 0] \cap I$ and $[0, \xi] \cap I$. Similarly, the bounds of f' are obtained by evaluating f' at the values $-\xi, -\zeta, 0, \zeta, \xi$ clipped to I.

NOTES

It is worth mentioning that some optimization could be done here, storing the active blobbies in an interval tree to accelerate the query of all candidates in every iteration of the interval refinement. However, due to the strategies described previously to limit the growth of $\mathcal{A}$, we have not confirmed in the implementation that the interval tree amortizes in practice.

Furthermore, when aggregating the bounds of ψ_i to determine whether $\phi(t, r(\cdot))$ may contain a root, we can discard the interval based on the fact that all ψ_i are nonnegative: if the lower bound of the partial sum is larger than $-T$, it will not recover from being bounded away from zero and therefore will not contain a root.

For a GPU implementation, the described approach of collecting the active set $\mathcal{A}$ per ray is not feasible because the number of elements is unbounded. However, the interval refinement approach can also be implemented by accumulating interval bounds on the fly in an `anyhit` program (see also [9]), adjusting $s_{\max}$ as described in Section 35.4.1. The strategy for rays inside the surface as described in Section 35.4.1 is possible in principle but requires some modifications to the stack to keep track of the minimum distance of all its elements.

35.5 RESULTS

We apply the ray tracing algorithm on two different data sets. The first scene is composed of 500 blobbies. They have been generated by randomizing their position within a unit radius sphere. Each particle comes with an initial velocity and acceleration. In Figure 35-12 we show how the BVH can be used to render higher-order motion blur in an efficient way and the benefit of having continuous derivatives all along the surface. We use the derivatives to apply the temporal antialiasing technique described by Tessari et al. [10]. The second asset is composed of 1,221,370 particles. It has been produced by simulating the explosion of a water bowl and surfacing the final result with an approach similar to that of Yu and Turk [12]. In Figure 35-13 we show how the anisotropy improves the shape of the thin lines of water produced by the explosion. We used Manuka [5] to run all our tests on a machine with 24 CPUs at a resolution of 1920×1080. In the first scene we had an average of 1.3 million rays per second, while for the second asset, where particles tend to overlap more to each other, we averaged 236,000 rays per second.

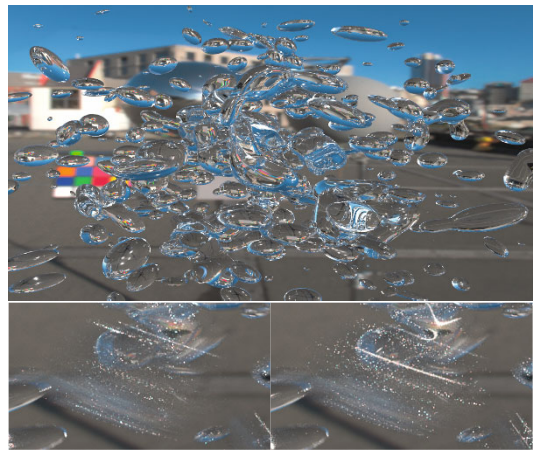


Figure 35-12. *Top: a static frame of our data set, composed of 500 anisotropic particles. Bottom: the details of one particle, after applying higher-order motion blur to it. On the right side, we exploited the derivatives to apply temporal antialiasing [10]. We limited the number of samples per pixel to 64, to show how the temporal antialiasing helps the convergence of the rendering.*

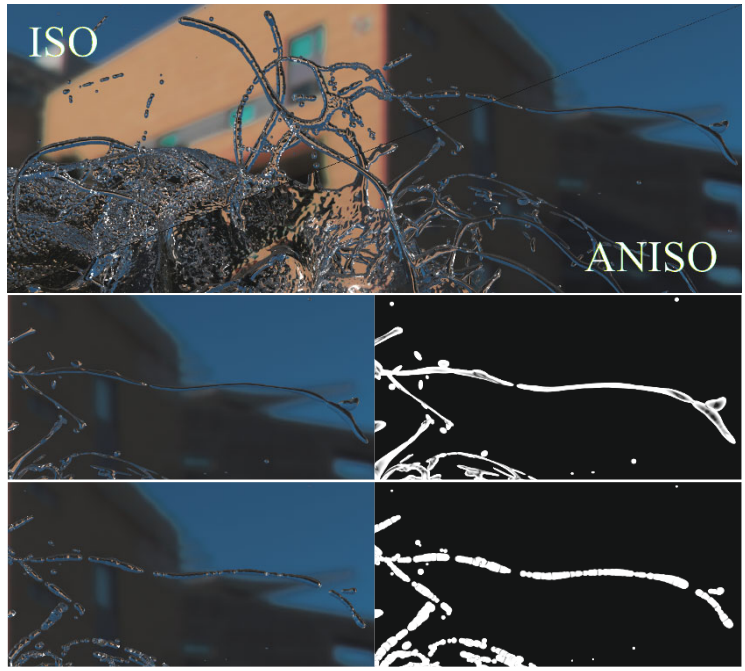


Figure 35-13. *In the case of a water bowl explosion, we have patterns of water representing thin walls and lines. Anisotropy helps in preserving the correct shape of these structures. In the top image, we can see how anisotropy compares to the isotropic case. The other images show the details of a thin line of water, with its derivatives on the right side.*

ACKNOWLEDGMENTS

We'd like to thank Louis-Daniel Poulin for his enormously helpful input in various discussions on special effects rendering.

REFERENCES

[1] Blinn, J. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, 1982. DOI: 10.1145/357306.357310.

[2] Capriani, O., Hvidegaard, L., Mortensen, M., and Schneider, T. Robust and efficient ray intersection of implicit surfaces. *Reliable Computing*, 6:9–21, 2000. DOI: 10.1023/A:1009921806032.

[3] Christensen, P. H., Fong, J., Laur, D. M., and Batali, D. Ray tracing for the movie 'Cars'. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 1–6, 2006. DOI: 10.1109/RT.2006.280208.

[4] Díaz, J. E. F. *Improvements in the Ray Tracing of Implicit Surfaces Based on Interval Arithmetic*. PhD thesis, Universitat de Girona, 2008.

[5] Fascione, L., Hanika, J., Leone, M., Droske, M., Schwarzhaupt, J., Davidovič, T., Weidlich, A., and Meng, J. Manuka: A batch-shading architecture for spectral path tracing in movie production. *ACM Transactions on Graphics*, 37(3):31:1–31:18, Aug. 2018. DOI: 10.1145/3182161.

[6] Kaplanyan, A. S., Hill, S., Patney, A., and Lefohn, A. Filtering distributions of normals for shading antialiasing. In *Proceedings of High Performance Graphics*, pages 151–162, 2016.

[7] Knoll, A. *Ray Tracing Implicit Surfaces for Interactive Visualization*. PhD thesis, School of Computing, Utah University, 2009.

[8] Museth, K. A flexible image processing approach to the surfacing of particle-based fluid animation (invited talk). In K. Anjyo, editor, *Mathematical Progress in Expressive Image Synthesis I: Extended and Selected Results from the Symposium MEIS2013*, pages 81–84. Springer Japan, 2014. DOI: 10.1007/978-4-431-55007-5_11.

[9] Singh, J. M. and Narayanan, P. J. Real-time ray-tracing of implicit surfaces on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):261–272, 2010. DOI: 10.1109/TVCG.2009.41.

[10] Tessari, L., Hanika, J., Dachsbacher, C., and Droske, M. Temporal normal distribution functions. In *Eurographics Symposium on Rendering—DL-only Track*, pages 1–12, 2020. DOI: 10.2312/sr.20201132.

[11] Tokuyoshi, Y. and Kaplanyan, A. S. Improved geometric specular antialiasing. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 8:1–8:8, 2019. DOI: 10.1145/3306131.3317026.

[12] Yu, J. and Turk, G. Reconstructing surfaces of particle-based fluids using anisotropic kernels. *ACM Transactions on Graphics*, 32(1):5:1–5:12, Feb. 2013. DOI: 10.1145/2421636.2421641.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 36

CURVED RAY TRAVERSAL

Christiaan Gribble
SURVICE Engineering

ABSTRACT

We present an implementation of curved ray traversal through volumes of spatially varying refractive indices. Our work is motivated by problems in gradient field tomography, including simulation, reconstruction, and visualization of unknown compressible flows. Reconstruction, in particular, requires computing, storing, and later retrieving sample points along each ray, which in turn necessitates a multi-pass traversal algorithm to overcome potentially burdensome memory requirements. The data structures and functions implementing this algorithm also enable direct visualization, including rendering of objects with varying refractive indices. We highlight a GPU implementation in OWL, the OptiX 7 Wrapper Library, and demonstrate sampling for reconstruction and interactive rendering of refractive objects. We also provide source code, distributed under a permissive open source license, to enable readers to explore, modify, or enhance our curved ray traversal implementation.

36.1 INTRODUCTION

Geometric optics, or sometimes *ray optics*, models light propagation as independent rays traveling through and interacting with different optical media according to a set of geometric principles. Here, a *ray* is simply a line or curve perpendicular to the propagation wavefront.

Geometric optics makes several assumptions about these rays to provide a straightforward but practical model of the underlying physics. In particular:

- Rays follow straight paths in a homogeneous medium.
- Rays follow curved paths in a medium with varying refractive indices.
- Rays bend (and, in some circumstances, split) at the interface between two different media.
- Rays can be scattered or absorbed by a medium as they propagate.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_36

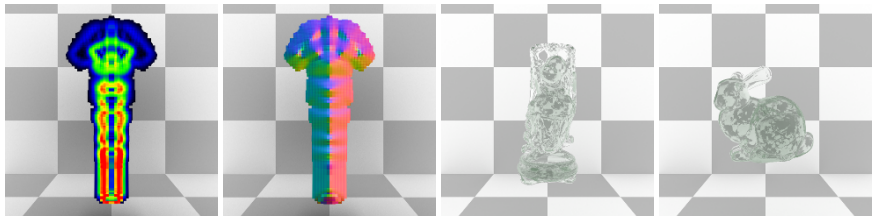


Figure 36-1. *Curved ray traversal. We present a multi-pass algorithm for curved ray traversal through volumes of spatially varying refractive indices, which we use for simulation, reconstruction, and visualization of unknown compressible flows (left images), as well as for interactive rendering of refractive objects (right images).*

Many computer graphics algorithms employ geometric optics to capture common illumination effects including image formation, reflection, and refraction. Here, we leverage geometric optics to implement curved ray traversal through volumes of spatially varying refractive indices. We apply this implementation to problems in gradient field tomography and interactive rendering.

We are exploring these areas to support near real-time decision-making in industrial imaging applications—including simulation, reconstruction, and visualization of unknown compressible flows. Reconstruction, in particular, requires computing, storing, and later retrieving sample points along each ray, which in turn necessitates a multi-pass traversal algorithm to overcome potentially burdensome memory requirements. The data structures and functions implementing this multi-pass algorithm also enable direct visualization, including interactive rendering of objects with varying refractive indices. Figure 36-1 depicts several images rendered by our implementation.

In the remainder of this chapter, we briefly review the physical basis of curved ray traversal to provide context for our algorithm and we highlight its utility in sampling for reconstruction and interactive rendering of refractive objects. We then review a GPU implementation of our multi-pass variant in OWL [24], the OptiX 7 Wrapper Library. We also provide the full source code [5], distributed under a permissive open source license, permitting readers to explore, modify, or enhance our foundational curved ray traversal implementation.

36.2 BACKGROUND

We leverage curved ray traversal to solve problems in gradient field tomography and interactive rendering. Importantly, our ability to interactively render high-fidelity images originates from the same physical principles governing gradient field tomography: the propagation of light as it travels through and interacts with different optical media.

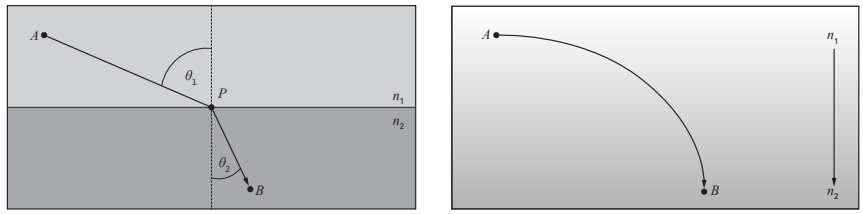
36.2.1 REFRACTION

Refraction in optical media is governed by *Fermat's principle* [3], which provides a link between geometric optics and wave optics. Loosely, Fermat's principle states that light traveling between two points follows the path of least time.

To illustrate, consider two media of different but homogeneous refractive indices, as in Figure 36-2a. Given a point A in the first medium and a point B in the second, the point P at which refraction occurs is the one that minimizes the time required for light to travel the path $\overrightarrow{AB}$. This phenomenon is expressed by the *law of refraction*, or *Snell's law*:

$$\frac{\sin\theta_2}{\sin\theta_1} = \frac{v_2}{v_1} = \frac{n_1}{n_2}, \tag{36.1}$$

where each θ_i is the angle measured from the boundary normal, v_i is the velocity of light in each medium, and n_i is the refractive index of each medium, respectively.



(a) *Refraction at an interface* **(b)** *Continuous refraction*

Figure 36-2. *Schematic illustration of Fermat's principle and the law of refraction. Fermat's principle states that light traveling between two points, A and B, follows the path of least time. (a) Given points A and B in two media of different but homogeneous refractive indices, point P is the point that minimizes the time taken by light to travel the path $\overrightarrow{AB}$. The law of refraction, or Snell's law, relates the angles θ_1 and θ_2 to the velocities v_1 and v_2 or, equivalently, to the refractive indices n_1 and n_2. (b) The path $\overrightarrow{AB}$ follows a curve in a medium for which the refractive index varies spatially, as in a compressible flow field.*

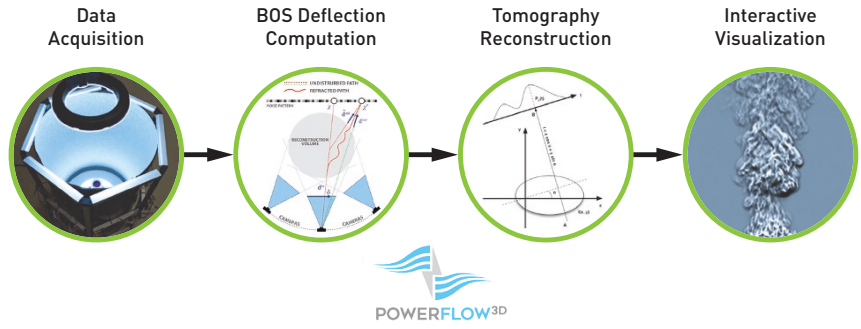


Figure 36-3. *The PowerFlow^{3D} pipeline. PowerFlow^{3D} is a prototype system for acquiring, processing, and visualizing 3D structure in experimental flows. PowerFlow^{3D} combines modern high-performance computing with existing methods for acquisition, reconstruction, and visualization of 3D flow features to help reveal critical information about the underlying structure of unknown flows.*

However, for a medium in which the refractive index varies spatially, the path $\overrightarrow{AB}$ itself changes continuously, as illustrated in Figure 36-2b. For such a medium, then, refracted light follows a curved path. In our context, this behavior manifests as light traverses a compressible flow field, where the refractive index at any point varies according to fluid density.

36.2.2 GRADIENT FIELD TOMOGRAPHY

Our multi-pass curved ray traversal algorithm supports PowerFlow^{3D} [6], an end-to-end prototype system for acquiring, reconstructing, and visualizing the underlying structure of unknown flows. PowerFlow^{3D} is based on work by Atcheson et al. [1], which describes both an apparatus for measuring refractive indices of compressible flows using an array of video cameras (i.e., *2D deflection sensing*) and a methodology for computing 3D refractive index gradient fields (i.e., *3D tomographic reconstruction*). The PowerFlow^{3D} pipeline is illustrated in Figure 36-3.

Both 2D deflection sensing and 3D tomographic reconstruction are based on the image formation process for background-oriented schlieren (BOS) methods [20], which are governed by continuous refraction for optically inhomogeneous media. In particular, the propagation of light in such media is described by the *ray equation of geometric optics*:

$$\frac{d}{ds}\left(n\frac{d\boldsymbol{x}}{ds}\right) = \nabla n, \tag{36.2}$$

where n is the refractive index field, $\boldsymbol{x}$ is the position of a photon traversing the ray, and ds is the differential path length along the ray. Several works in the computer graphics literature—for example, those authored by Stam and Langénou [22], Gutierrez et al. [8], and Ihrke et al. [11]—leverage the ray equation reformulated as a system of coupled first-order ordinary differential equations to describe curved ray paths:

$$n\frac{d\boldsymbol{x}}{ds} = \boldsymbol{d}, \quad \frac{d\boldsymbol{d}}{ds} = \nabla n. \tag{36.3}$$

In this formulation, $\boldsymbol{d}$ describes the local ray direction scaled by the local refractive index. Integrating this equation relates the refractive index gradient to 3D ray deflections in tomographic reconstruction:

$$\boldsymbol{d}^{\text{out}} = \int_c \nabla n \, ds + \boldsymbol{d}^{\text{in}}, \tag{36.4}$$

where c is the curved ray path and $\boldsymbol{d}^{\text{in}}$ and $\boldsymbol{d}^{\text{out}}$ denote the incoming and outgoing ray directions with respect to the medium.

Importantly, the ray equation naturally incorporates straight and curved ray paths to simulate phenomena like reflection and refraction—phenomena with significant impact on correctness in gradient field tomography and on visual fidelity in rendering.

For tomographic reconstruction, the unknown vector-valued function ∇n is discretized by a set of normalized basis functions Φ_i with unknown coefficient vectors $\boldsymbol{n}_i$:

$$\widehat{\nabla n} = \begin{pmatrix} \sum_i n_i^x \Phi_i \\ \sum_i n_i^y \Phi_i \\ \sum_i n_i^z \Phi_i \end{pmatrix} = \sum_i \boldsymbol{n}_i \Phi_i. \tag{36.5}$$

This formulation relates 3D ray deflections to 3D refractive index gradients as

$$\int_c \nabla n \, ds = \int_c \sum_i \boldsymbol{n}_i \Phi_i \, ds = \sum_i \boldsymbol{n}_i \int_c \Phi_i \, ds = \boldsymbol{d}^{\text{out}}_{(x,y,z)} - \boldsymbol{d}^{\text{in}}_{(x,y,z)}, \tag{36.6}$$

where $\boldsymbol{n}_i = (n_i^x, n_i^y, n_i^z)$ is a three-component coefficient vector independently parameterizing the three gradient components in each dimension. The discretization results in a system of linear equations over curved ray paths c, one for each gradient component:

$$\boldsymbol{S}\boldsymbol{n}_{(x,y,z)} = \boldsymbol{d}^{\text{out}}_{(x,y,z)} - \boldsymbol{d}^{\text{in}}_{(x,y,z)}. \tag{36.7}$$

Solving these systems for the coefficient vectors $\boldsymbol{n}_i$ recovers the refractive index gradient field $\widehat{\nabla n}$.

1 Compute deflection vectors $\boldsymbol{d}^{\text{out}}_{(x,y,z)} - \boldsymbol{d}^{\text{in}}_{(x,y,z)}$;
2 Set initial estimate $\widehat{\nabla n} = \mathbf{0}$;
repeat
3 Compute curved ray paths c;
4 Construct matrix $\boldsymbol{S}$;
5 Solve linear system $\boldsymbol{S}\boldsymbol{n}_{(x,y,z)} = \boldsymbol{d}^{\text{out}}_{(x,y,z)} - \boldsymbol{d}^{\text{in}}_{(x,y,z)}$;
until *convergence*;
6 Integrate $\widehat{\nabla n}$ to recover n;

Figure 36-4. *Gradient field tomography: reconstruction.*

Typically, these steps are repeated in an iterative manner until some termination condition is met. Finally, integrating the gradient field $\widehat{\nabla n}$ yields the refractive index field n. This process is analogous to computing a surface from normal vectors using a discretized form of the Laplacian operator,

$$\Delta n = \nabla \cdot \widehat{\nabla n}. \tag{36.8}$$

Here, the left-hand side is discretized, whereas the right-hand side is computed using the recovered $\widehat{\nabla n}$, and the resulting Poisson equation is solved for n.

The reconstruction algorithm is outlined in Figure 36-4. High-performance and memory-efficient curved ray traversal (Figure 36-4, step 3) is a critical component of the overall reconstruction process.

We use Eigen [7, 12] to implement numerical processing within the PowerFlow^{3D} pipeline—the linear system solution and Poisson integration processes supporting tomographic reconstruction, specifically. Eigen is a library of C++ template headers for linear algebra, matrix and (mathematical) vector operations, and numerical solvers. Eigen exploits vector processing with SIMD operation on modern CPUs, avoids dynamic memory allocation, unrolls loops when possible, and pays special attention to cache-friendliness for large matrices. Together, these characteristics make Eigen an ideal library for prototyping the numerical processing required by PowerFlow^{3D}.

Tractability of reconstruction relies on a sparse formulation of matrix $\boldsymbol{S}$. In particular, the entries of matrix $\boldsymbol{S}$ consist of line integrals over basis

functions Φ_i:

$$\boldsymbol{S} = \begin{pmatrix} \int_{c_1} \Phi_1 \, ds & \int_{c_1} \Phi_2 \, ds & \cdots & \int_{c_1} \Phi_{N_b} \, ds \\ \int_{c_2} \Phi_1 \, ds & \int_{c_2} \Phi_2 \, ds & \cdots & \int_{c_2} \Phi_{N_b} \, ds \\ \vdots & \vdots & \ddots & \vdots \\ \int_{c_{N_m}} \Phi_1 \, ds & \int_{c_{N_m}} \Phi_2 \, ds & \cdots & \int_{c_{N_m}} \Phi_{N_b} \, ds \end{pmatrix}, \tag{36.9}$$

where N_m is the number of deflection measurements across all views simultaneously and N_b is the number of basis functions. The matrix $\boldsymbol{S}$ is thus quite large: $N_m \times N_b$ entries, with N_m = width $\times$ height $\times N_v$ for width $\times$ height images and N_v views, and $N_b = N_x \times N_y \times N_z$ voxels in the reconstruction volume.

Assuming a dense, 64-bit double-precision floating-point format, our fidelity goals dictate that matrix $\boldsymbol{S}$ alone would consume several petabytes of memory, thereby overwhelming even typical distributed-memory supercomputing platforms. Following Atcheson et al. [1], we thus use basis functions with finite support to impose sparsity on matrix $\boldsymbol{S}$.

In particular, we use radially symmetric linear basis functions $\Phi_i = \max(0, 1 - r)$ for radius r. These functions preserve sparseness while allowing interpolation in the 3D solution space. Basis functions are aligned to a regular grid, excluding those with support completely outside a conservative visual hull. This so-called *visual hull restricted tomography* was introduced for flame reconstruction [10] and is necessary to reconstruct refractive index gradients from a sparse set of input views with reasonable resource requirements.

Given basis functions Φ_i, we then carefully and efficiently construct and solve the resulting sparse linear system (Figure 36-4, steps 4–5) using Eigen's sparse least-squares conjugate gradient (LSCG) numerical solver. Although Eigen exploits only SIMD operation within any particular invocation of the LSCG solver, we leverage multithreaded processing in step 5 by executing three solver instances—one for each gradient component in (x, y, z)—across multiple CPUs simultaneously.

Likewise, we integrate the reconstructed gradient field $\widehat{\nabla n}$ to compute the refractive index field n by solving a Poisson equation (Figure 36-4, step 6) using Eigen's supernodal lower-upper (LU) factorization solver. Here, too, Eigen exploits SIMD operation within the solver, but unlike the linear system solution in step 5, Poisson integration does not afford opportunities to

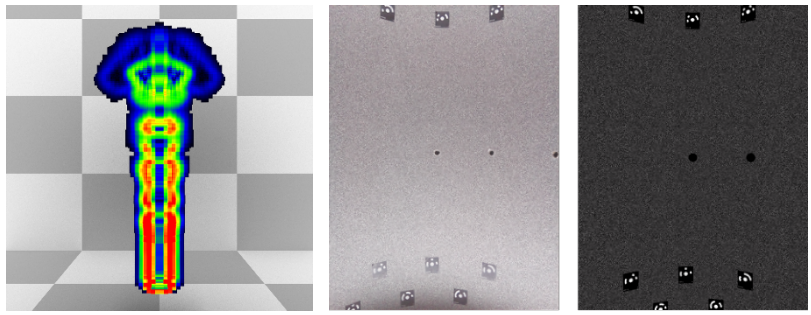
leverage even explicit multithreaded operation. Nevertheless, the overall runtime of our current PowerFlow^{3D} prototype is limited by the linear system solution in step 5, rather than curved ray traversal in step 3, matrix construction in step 4, or even Poisson integration in step 6. Interested readers are referred to the work by Gribble et al. [6] for additional details regarding PowerFlow^{3D}.

36.2.3 INTERACTIVE RENDERING

Our multi-pass curved ray traversal algorithm supports not only tomographic reconstruction but also direct visualization of 3D refractive index gradient fields.

Within PowerFlow^{3D}, we exploit this functionality for both interactive visualization and high-fidelity simulated data capture. In the former, we use direct visualization to reveal the underlying structure of complex flows, as in Figure 36-5a. In the latter, we use the data capture simulator both to generate ground-truth data for reconstruction-related research and development tasks and to qualitatively assess discrepancies between our physical data acquisition apparatus and its ideal virtual counterpart, as in Figure 36-5b.

At the same time, interactive rendering of objects with complex optical properties remains a central theme of realistic image synthesis. As noted by



(a) *Interactive visualization* **(b)** *Simulated data capture*

Figure 36-5. *Direct visualization of 3D refractive index gradient fields. In addition to tomographic reconstruction, we leverage curved ray traversal for both interactive visualization and high-fidelity simulated data capture in PowerFlow^{3D}. (a) Direct visualization reveals the underlying structure of complex flow fields, for example, using pseudo-color mappings of gradient magnitude, as shown here. (b) Direct visualization also supports high-fidelity simulated data capture, which is used to generate ground-truth data for development, debugging, and validation tasks; here, comparing an actual capture image (left) to the corresponding image generated by our simulator (right) enables qualitative assessment of potential alignment discrepancies in the physical apparatus.*

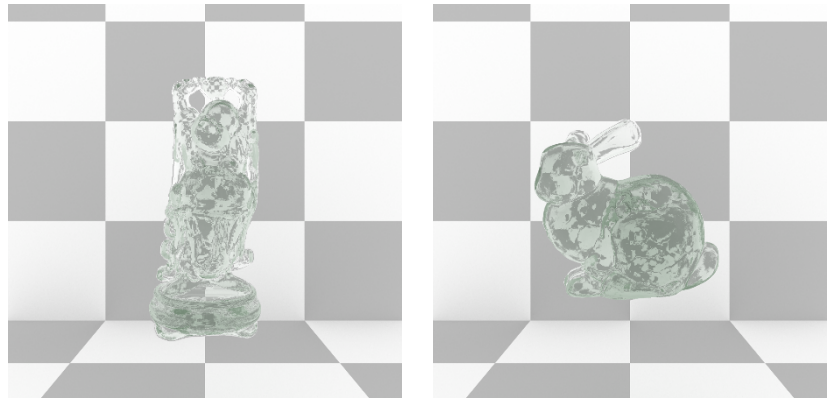


Figure 36-6. *Interactive rendering of refractive objects. Our ability to simulate the visual qualities of objects with complex optical properties originates from the same physical principles governing gradient field tomography: the propagation of light as it travels through and interacts with different optical media. Here, we use the same application supporting direct visualization in PowerFlow^{3D} to interactively render refractive objects with full path tracing.*

Ihrke et al. [11], the interaction of light with different optical media induces the striking visual beauty of our environment:

> [This beauty] has its physical origin in the interplay of the involved light/matter interaction processes that take place while light passes material boundaries, while it travels through the interior of an object, and even while it interacts with the object's surroundings.

As illustrated in Figure 36-6, the same application used for direct visualization in PowerFlow^{3D} enables interactive rendering of refractive objects, in this case using path tracing [13]. Although only primary visibility is required for tomographic reconstruction, our algorithm supports path tracing and thus integrates with typical ray-based renderers. As discussed in Section 36.3, we represent the volume extent as a scene object using a cube as the *bounding* (or *proxy*) *geometry*. The cube itself comprises 12 triangle primitives, which exploits hardware acceleration for initial ray/volume intersection queries on RTX-enabled GPUs [16].

Curved ray traversal—and the multi-pass algorithm that we describe in the next section—captures the complex interactions of light with different optical media and thus plays a critical role in both gradient field tomography and interactive rendering.

36.3 IMPLEMENTATION

We implement our multi-pass curved ray traversal algorithm using OWL [24], the OptiX 7 Wrapper Library. Recently, many high-performance graphics application programming interfaces (APIs), including NVIDIA's OptiX, have shifted toward providing lower-level control of graphics resources [14]. This shift allows experienced developers to retain full control while still leveraging the benefits of highly optimized APIs—albeit at the cost of additional application-level complexity.

In contrast, OWL wraps the new CUDA-centric OptiX 7 API in a higher-level interface to provide the convenience of the classic OptiX API. OWL makes porting existing OptiX applications easier and aids developers in getting new OptiX 7 applications running quickly to enable easy experimentation.

We use OWL as the basis for our curved ray traversal implementation. Though OWL is required to build our example application from source, the supporting device-side data structures and ray tracing programs themselves can be adapted to other GPU- or CPU-based frameworks that provide similar ray tracing abstractions. We highlight our implementation components in the remainder of this section.

36.3.1 OVERVIEW

As noted, tomographic reconstruction requires computing, storing, and retrieving sample points along curved ray paths. The number of samples along any one ray is not known a priori, particularly because the paths become increasingly accurate from one iteration of the reconstruction algorithm to the next. To avoid dynamic memory allocation on the device (which is potentially expensive), and to avoid intentionally overprovisioning memory resources (which is potentially wasteful and does not necessarily cover all use cases), we instead employ a preallocated fixed-size buffer for each ray when collecting sample points and implement a multi-pass algorithm, as outlined in Figure 36-7.

In particular, we allocate memory based on a user-controlled runtime parameter specifying the maximum per-ray sample count for a single pass. If any ray requires more samples to complete traversal, we execute another pass—there is no sharing of unused buffer entries across rays within a pass. Although only unfinished rays actually execute subsequent passes, this straightforward but practical approach does incur the overheads associated with pass setup/teardown, whether there are many or few such unfinished rays.

Host-Side Processing

```
1  Initialize samples, done buffers;
2  pass ← 0;
3  Launch device-render(pass, . . .);
4  Copy samples buffer;
5  n_complete ← sum-reduce(done);
6  while n_complete < n_total do
7      pass ← pass + 1;
8      Launch device-render(pass, . . .);
9      Copy samples buffer;
10     n_complete ← sum-reduce(done);
```

Device-Side Processing

```
1  if pass = 0 then
2      Initialize curved ray paths c;
3      done_c ← trace-path(c, . . .);
4  else
5      if done_c then
6          return;
7      Load curved ray paths c;
8      done_c ← resume-path(c, . . .);
```

Figure 36-7. *Multi-pass curved ray traversal.*

This approach is not intended to balance samples across rays, but to ensure that traversal always completes while respecting memory constraints. Though more sophisticated memory-management schemes could be used, sampling is not the current bottleneck in PowerFlow^{3D}, and whether or not the complexities of such schemes eliminate enough of the subsequent passes to outperform our current approach remains unclear. These and similar questions make for an interesting performance study, but this avenue of exploration remains as future work.

Critical to our multi-pass variant is the ability to postpone (and later resume) a ray mid-flight—in particular, when its sample buffer is full. We first highlight the core data structures supporting our multi-pass algorithm in Section 36.3.2, and then we highlight the OptiX 7 ray tracing programs and helper functions that leverage these data structures to implement the device-side processing in Section 36.3.3.

In contrast to reconstruction sampling, curved ray traversal for interactive rendering does not require multiple passes—sample points are not stored for later processing, so efficiency concerns related to memory management are less acute in this context. However, the data structures and ray tracing programs we describe in the remainder of this section also enable interactive rendering, and thus provide a high-performance, memory-efficient foundation for applications requiring both sampling and direct visualization, including our PowerFlow^{3D} prototype.

In practice, we use a combination of static and dynamic polymorphism to invoke the necessary variant of our curved ray traversal algorithm: types and behaviors are specialized for sampling, where necessary.[1] These differences are most clearly evident in the ray tracing programs discussed in Section 36.3.3, though some data structures include elements that support only reconstruction sampling and are not used for interactive rendering. A more highly optimized version of curved ray traversal is, in theory, possible with an implementation supporting only interactive rendering. Our multi-pass implementation provides a basis for this possibility, but we focus on the elements that support sampling for reconstruction in the discussion that follows. Even so, our implementation achieves over 12 million samples per second (or nearly 7 frames per second) with volumes up to $224 \times 224 \times 224$ voxels and framebuffers up to 768×768 pixels in resolution using a single NVIDIA Quadro RTX 8000 GPU. Additional optimizations targeting *interactive-rendering-only* use cases are left as an exercise to the reader.

36.3.2 CORE DATA STRUCTURES

We now outline the core data structures used to implement the multi-pass algorithm in Figure 36-7; the OptiX 7 ray tracing programs implementing device-side processing are highlighted in Section 36.3.3.

In the code listings that follow, we include only the most relevant data members; some data members (and all member function declarations) are omitted for brevity. The full source code for these and other implementation elements is available online.

`VOLUMEGEOMDATA` STRUCTURE

We encode refractive index gradients in a 3D rectilinear grid of (x, y, z) triples, `VolumeGeomData`. This structure also encodes auxiliary information used throughout traversal. The `VolumeGeomData` structure definition is shown in Listing 36-1.

We represent the volume extent as a scene object using a cube as the proxy geometry; the cube itself comprises 12 triangle primitives. This triangle-based representation not only exploits hardware acceleration on

[1] In fact, our source code distribution supports a third *color-mapping* variant that uses multiple passes to first gather data about volume sampling and then map particular elements from this data to pixel colors. Like reconstruction sampling, this color-mapping multi-pass algorithm variant is implemented using both static and dynamic polymorphism, but we ignore these details in this discussion for clarity.

Listing 36-1. *VolumeGeomData structure.*

```
struct VolumeGeomData {
  box3f bounds;
  vec3i dims;

  vec3f* vertex;
  vec3i* index;

  vec3f* gradient;
  int*   ior_mask;
};
```

RTX-enabled GPUs, but also allows for extensions to our current implementation with arbitrary bounding geometries [22, 11, 23], which can give tighter ray/volume traversal bounds for interactive rendering. An OptiX 7 closest-hit program translates ray/triangle intersection information to a volume entry point and initiates volume traversal, as described in Section 36.3.3.

TRAVERSALDATA STRUCTURE

To postpone and later resume path traversal, we must store and later load the traversal-related state for incomplete ray paths between each pass of our algorithm. This state is encoded in the per-path `TraversalData` structure shown in Listing 36-2. The `TraversalData` structure includes both the general traversal state and elements specific to volume traversal. These device-side components are initialized or loaded at the beginning of each pass and updated throughout traversal, as described in Section 36.3.3.

Listing 36-2. *TraversalData structure.*

```
struct TraversalData {
  Random random;

  int pidx;
  int depth;

  Interaction::Event event;

  vec3f org;
  vec3f dir;
  vec3f atten;

  const VolumeGeomData* vptr;

  float eta;
  int   ndata;
  float distance;
};
```

PERRAYDATA STRUCTURE

Additional per-ray data, such as path attenuation and traversal outputs, is stored in the `PerRayData` structure shown in Listing 36-3. This structure also holds references to the corresponding `TraversalData` and random number generator for convenience. Unlike `TraversalData`, which persists from one pass to the next, a new device-side `PerRayData` object is initialized with each pass of our algorithm.

Listing 36-3. *`PerRayData` structure.*

```
struct PerRayData {
  TraversalData& tdata;
  Random&        random;

  vec3f atten;

  struct {
    Path::Status status;
    vec3f org;
    vec3f dir;
    vec3f atten;
  } out;
};
```

SAMPLEDATA STRUCTURE

The *samples* buffer in Figure 36-7 stores `SampleData` elements; this structure is defined in Listing 36-4. As can be seen, each `SampleData` entry comprises a 3D sample position and, in our current implementation, a count of the refractive index gradient elements encountered during volume traversal (which is used in direct visualization). Though simple, this data structure can be extended to store additional per-sample data, if necessary; we found this abstraction to be useful during initial debugging, for example, as it enabled collection of an additional sample-related state with only local source code modifications.

Listing 36-4. *`SampleData` structure.*

```
struct SampleData {
  vec3f pt;
  int   ndata;
};
```

For each pass and for each ray path, at most $N_S \leq$ `MaxSampleCount` samples are stored, where `MaxSampleCount` is the user-controlled maximum per-ray sample count in each pass. For the duration of a width $\times$ height frame, the

samples buffer comprises width × height × `MaxSampleCount` entries to avoid device-side dynamic allocation. Though overprovisioning may still occur (`MaxSampleCount` may still exceed the actual number of samples required by any ray path), users retain full runtime control over `MaxSampleCount`—and, thus, over the memory requirements necessary to support any particular invocation of curved ray traversal. Runtime control not only permits adjustments throughout the application lifetime, but also allows users to trade processing time for memory and vice versa.

Any ray path requiring N_s > `MaxSampleCount` samples is postponed when its entry in the *samples* buffer is full—that is, when for path c, N_s = `MaxSampleCount` in the current pass—and thus induces an additional pass. These details are described more fully in Section 36.3.3.

OTHER ELEMENTS

The *samples* buffer in Figure 36-7 communicates device-side results to the host for additional processing, if required. For example, in the case of sampling for reconstruction, per-pass samples populate host-side data structures that are later passed to downstream processing. In our current implementation, we store complete ray paths—not just the within-volume sample points but also the initial origin and all ray/primitive intersection points along the path—throughout traversal. The host-side processing discards non-volume samples as it populates host-side data structures for downstream reconstruction operations, as required.

The *done* buffer, also shown in Figure 36-7, plays a critical role: this buffer communicates path traversal progress from device to host. In particular, the *done* buffer stores `int` elements, each of which acts as a flag indicating whether or not the corresponding ray path c has been traced to completion. Though more compact representations of these flags could be used—for example, just a single bit per path—we opt for `int` values as a straightforward way to encode these flags on the device. Each path flag, $done_c$, is initialized or loaded in the *complete* state ($done_c \leftarrow 1$), and the flag is set to the *incomplete* state ($done_c \leftarrow 0$) only if the corresponding path is postponed mid-flight, as discussed in Section 36.3.3. Sum-reduction over the *done* buffer computes the number of paths completed thus far and serves as the termination condition for our multi-pass algorithm. Sum-reduction is currently implemented host-side using OpenMP [4, 18], though device-side implementation—for example, using CUDA directly or using *Thrust*, the CUDA C++ template library [2, 17]—is also possible.

The example application demonstrating curved ray traversal leverages several other data structures, including elements supporting direct visualization and results validation, as well as elements typical of any modern ray tracing application (framebuffers, accumulation buffers, windowing components, and so forth). We omit discussion of these data structures for brevity; for additional information, the interested reader is instead referred to the many available resources describing modern ray tracing generally [21, 9, 19] and ray tracing with OWL and OptiX 7 specifically [24, 14, 15], as well as to the application source code itself.

36.3.3 RAY TRACING PROGRAMS

The OptiX 7 API gives a client application full control over many operations that were managed by the OptiX library in previous versions of the API. OWL, the OptiX 7 Wrapper Library, hides some of these low-level OptiX 7 operations behind a clean and easy-to-use higher-level interface reminiscent of the classic OptiX API. We use OWL as the basis of our curved ray traversal implementation.

With OptiX, device code is organized into several *ray tracing programs*, which together with the internal OptiX scheduling algorithms and bounding volume hierarchy (BVH) traversal programs, comprise a full *ray tracing kernel*. We highlight the ray tracing programs and critical device-side helper functions supporting our implementation in this section. We assume a working knowledge of the OptiX 7 programming model in this discussion, but Morley [14] provides an in-depth review of the OptiX 7 ray tracing pipeline for interested readers.

We implement our multi-pass algorithm within a brute-force path tracing kernel that supports progressive rendering to accumulate pixel samples over successive frames. The path tracer currently supports Lambertian surfaces and spatially varying refractive volumes with Beer's Law attenuation, but can be extend with support for additional material models in a straightforward manner. Buffers and parameters supporting or controlling runtime behavior, including the framebuffer; accumulation buffer; the *samples*, *done*, and `TraversalData` data buffers; and the current *render mode* and *pass* are communicated through a `LaunchParams` structure named `optixLaunchParams`, referenced throughout the code snippets in this section. Please refer to the source code distribution for details of these and other elements, as required.

RAY GENERATION PROGRAM

The *ray generation program* provides an entry point to device-side OptiX 7 ray tracing kernels; in our context, ray generation is invoked by launching the **device-render** kernel in Figure 36-7.

As can be seen in Listing 36-5, the actual ray generation program simply dispatches `rayGenFcn::run`, a member function of a class templated over an enumerated type corresponding to the current render mode. We provide two `rayGenFcn` variants: one for interactive rendering and one for reconstruction sampling. For interactive rendering, the function simply maps its launch index to a random sample within the corresponding pixel, traces a ray from the camera through that sample, and accumulates the resulting path attenuation.

Listing 36-5. *OptiX 7 Ray generation program.*

```
OPTIX_RAYGEN_PROGRAM(rayGen)() {
  switch (optixLaunchParams.rmode) {
  case Render::Mode::Normal:
    rayGenFcn<Render::Mode::Normal>::run();
    break;

  case Render::Mode::Sample:
    rayGenFcn<Render::Mode::Sample>::run();
    break;
  }
}
```

The latter variant, shown in Listing 36-6, implements the device-side processing in Figure 36-7: for each pass, we initialize or load ray paths and then trace or resume path traversal, as appropriate. The helper functions `tracePath` and `resumePath` implement these operations, as discussed next.

Listing 36-6. *`rayGenFcn<Render::Mode::Sample>` class template specialization.*

```
class rayGenFcn<Render::Mode::Sample> {
public:

  static __device__ void run() {
    const RayGenData& self    = owl::getProgramData<RayGenData>();
    const vec2i       pixelID = owl::getLaunchIndex();

    const vec2i& fbSize  = optixLaunchParams.fbSize;
    const int    pidx    = pixelID.y*fbSize.x + pixelID.x;

    const int& pass = optixLaunchParams.pass;
    if (pass == 0) {
      int&           nsamples = optixLaunchParams.nsamplesBuffer[pidx];
      int&           done     = optixLaunchParams.doneBuffer    [pidx];
      TraversalData& tdata    = optixLaunchParams.tdataBuffer   [pidx];
      PerRayData     prd(tdata);
```

```
17
18        // Clear values and ...
19        tdata = TraversalData(pidx);
20        tdata.random.init(pixelID.x, pixelID.y);
21
22        nsamples = 0;
23        done     = 1;
24
25        // ... trace ray.
26        const vec2f screen = (vec2f(pixelID) +
27                              vec2f(0.5f, 0.5f))/vec2f(fbSize);
28
29        const vec3f org = self.camera.origin;
30        const vec3f dir
31          = self.camera.lower_left_corner
32          + screen.u*self.camera.horizontal
33          + screen.v*self.camera.vertical
34          - self.camera.origin;
35
36        Ray ray(org, normalize(dir), T_MIN, T_MAX);
37        tracePath<Render::Mode::Sample>(self, ray, prd);
38      }
39      else {
40        int& done     = optixLaunchParams.doneBuffer    [pidx];
41        int& nsamples = optixLaunchParams.nsamplesBuffer[pidx];
42        if (done) {
43          nsamples = 0;
44          return;
45        }
46
47        // Load traversal data.
48        TraversalData& tdata = optixLaunchParams.tdataBuffer[pidx];
49
50        Ray        ray(tdata.org, tdata.dir, T_MIN, T_MAX);
51        PerRayData prd(tdata);
52
53        // Clear values and ...
54        nsamples = 0;
55        done     = 1;
56
57        // ... resume tracing.
58        resumePath(self, ray, prd);
59      }
60    }
61  };
```

TRACEPATH HELPER FUNCTION

The `tracePath` helper function in Listing 36-7 implements the iterative path tracing loop in step 3 of Figure 36-7: for each new ray, the corresponding `TraversalData` and `PerRayData` elements are updated and the sample point is stored. If the *samples* buffer is full, the ray is postponed and the function returns. If the sample is stored successfully, however, we invoke the `owl::traceRay` function to execute the OptiX 7 ray tracing pipeline, including

acceleration structure traversal, scene geometry intersection, and closest-hit processing.

Listing 36-7. *tracePath helper function.*

```
template<Render::Mode Mode>
inline __device__
vec3f tracePath(const RayGenData& rgd, Ray& ray, PerRayData& prd) {
  TraversalData& tdata = prd.tdata;
  tdata.org = ray.origin;
  tdata.dir = ray.direction;

  vec3f& atten = prd.atten;
  int&   depth = tdata.depth;

  int MaxDepth = MAX_DEPTH;

  while (depth < MaxDepth) {
    if (Mode == Render::Mode::Sample) {
      if (!storeSample(prd))
        return vec3f(0.f);
    }

    owl::traceRay(rgd.world, ray, prd);

    if (prd.out.status == Path::Status::Cancelled ||
        prd.out.status == Path::Status::Postponed)
      return vec3f(0.f);
    else if (prd.out.status == Path::Status::Missed) {
      atten *= missColor(ray);
      return atten;
    }
    else if (prd.out.status == Path::Status::Bounced)
      atten *= prd.out.atten;

    // Trace another ray.
    const vec3f& org = prd.out.org;
    const vec3f  dir = normalize(prd.out.dir);

    ray = Ray(org, dir, T_MIN, T_MAX);

    tdata.org = org;
    tdata.dir = dir;

    prd.out.status = Path::Status::Invalid;

    ++depth;
  }

  return atten;
}
```

Once complete, these downstream OptiX pipeline operations return control to `tracePath`, which determines the next appropriate action based on the path traversal outputs:

- `Path::Status::Cancelled` indicates that the path has been cancelled; in our current implementation, only non-primary rays interacting with the volume geometry are cancelled, which simulates fully opaque shadowing for volume proxy geometries. However, when extended to support other material models, rays could also be cancelled as a result of BRDF sampling.

- `Path::Status::Postponed` indicates that the path requires N_s > `MaxSampleCount` sample points; in this case, storing the (`MaxSampleCount` + 1)-th sample fails, so path traversal is postponed and induces an additional pass.

- `Path::Status::Missed` indicates that the path has escaped to the environment; in this case, the corresponding path attenuation is updated and returned to the caller.

- `Path::Status::Bounced` indicates that the ray has intersected valid scene geometry and should continue propagation. In this case, the corresponding *closest-hit* processing has recorded data about the interaction event—and in the case of our volume geometry, completed curved ray traversal—so the path attenuation is updated and control falls through to trace another ray segment for this path.

This process continues until either the current path segment undergoes an event that returns control to the caller (cancelled, postponed, or missed) or the maximum path tracing depth, `MAX_DEPTH`, is reached. In the latter case, the path is terminated and the corresponding path throughput is returned to the caller.

`RESUMEPATH` HELPER FUNCTION

The `resumePath` helper function in Listing 36-8 implements functionality to resume a postponed ray path in step 8 of Figure 36-7: each such path's state is restored from the `TraversalData` state stored in the previous pass, and path tracing continues based on that state.

The appropriate subsequent actions are determined by the path interaction event tracked and stored in the previous pass:

- `Interaction::Event::Exit` indicates that the path was postponed just after completing volume traversal; in this case, control simply falls through to the `tracePath` function to continue propagation.

Listing 36-8. *`resumePath` helper function.*

```
inline __device__
void resumePath(const RayGenData& rgd,
                      Ray&        ray,
                      PerRayData& prd) {
  if (prd.tdata.event == Interaction::Event::Exit) {
    // Postponed from previous traverse(...) call but after exiting
    //   volume, so just fall through to continue tracing.
  }
  else if (prd.tdata.event == Interaction::Event::Miss) {
    // Postponed when attempting to store path end point, so store
    //   endpoint and return.
    storeSample(prd);
    return;
  }
  else if (prd.tdata.event == Interaction::Event::Traverse) {
    // Postponed mid-traversal from previous traverse(...) call, so
    //   resume traversal.
    const VolumeGeomData& volume = *(prd.tdata.vptr);
    traverse<Render::Mode::Sample>(volume, prd);

    if (prd.out.status == Path::Status::Postponed) {
      // Did not finish traversing volume, so return to caller
      //   (through rayGen()) for another pass.
      return;
    }

    // Finished traversing volume, so initialize next ray, increment
    //   depth, and ...
    ray = Ray(prd.tdata.org, prd.tdata.dir, T_MIN, T_MAX);
    prd.out.status = Path::Status::Invalid;
    ++prd.tdata.depth;

    // ... fall though to continue tracing.
  }

  // Continue propagation.
  tracePath<Render::Mode::Sample>(rgd, ray, prd);
}
```

> `Interaction::Event::Miss` indicates that the path was postponed when attempting to store the path endpoint after a miss event in the `tracePath` function; in this case, we simply store the path endpoint and return control to the caller—the corresponding path is complete.

> `Interaction::Event::Traverse` indicates that the path was postponed during volume traversal; in this case, the `VolumeGeomData`-specific elements of `TraversalData` are valid and encode the state of the path within the corresponding volume object. As a result, we invoke the volume `traverse` helper function directly to continue traversal. Once this function returns, we check the path traversal status, as the path may

have been postponed once again; if so, the path state has been updated appropriately and we simply return to the caller. If, however, volume traversal completed successfully, we initialize the next ray segment and controls falls through to the `tracePath` function to continue propagation.

Together, the `tracePath` and `resumePath` implement the core path tracing loop within the context of Figure 36-7. Whether we initiate path traversal via primary rays or resume traversal from a previous pass, we eventually invoke the `tracePath` helper function, which in turn calls the `owl::traceRay` function to execute the OptiX 7 ray tracing pipeline.

In this pipeline, acceleration structure traversal and triangle-based scene geometry intersection are encapsulated within the OptiX 7 runtime, and the pipeline includes hardware acceleration on RTX-enabled GPUs.

In contrast, closest-hit processing is exposed to the user; we leverage a closest-hit program specific to our volume scene geometry to implement the propagation of light through the volume according to the ray equation of geometric optics, as described in Section 36.2. This program and the supporting helper functions are described next.

CLOSEST-HIT PROGRAM

After ray traversal and geometry intersection, the *closest-hit program* executes for the nearest intersection point along each ray, if any. Closest-hit programs typically calculate values derived from ray/primitive intersection data, perform shading operations (where applicable), and pass results back to the ray generation program.

In our context, the closest-hit program for our volume object simply dispatches `intersect`, a function templated over the current render mode, much like the ray generation program. Unlike ray generation, however, this program does not require specialization—it simply translates ray/triangle intersection data to a volume entry point, initializes the appropriate members of the corresponding `TraversalData` structure, and invokes volume traversal. The volume closest-hit program and `intersect` function template are shown in Listing 36-9.

`TRAVERSE` HELPER FUNCTION

Recall that curved ray traversal within volumes of spatially varying refractive indices is governed by the ray equation of geometric optics, as discussed in

Listing 36-9. *OptiX 7 closest-hit program.*

```
OPTIX_CLOSEST_HIT_PROGRAM(Volume)() {
  PerRayData& prd = owl::getPRD<PerRayData>();

  switch (optixLaunchParams.rmode) {
  case Render::Mode::Normal:
    intersect<Render::Mode::Normal>(prd);
    break;

  case Render::Mode::Sample:
    intersect<Render::Mode::Sample>(prd);
    break;
  }
}

template<Render::Mode Mode>
inline __device__
void intersect(PerRayData& prd) {
  // Compute hit point.
  const vec3f org  = optixGetWorldRayOrigin();
  const vec3f dir  = optixGetWorldRayDirection();
  const float thit = optixGetRayTmax();
  const vec3f hitP = org + thit*dir;

  // Prepare traversal data.
  const VolumeGeomData& self  = owl::getProgramData<VolumeGeomData>();
        TraversalData&  tdata = prd.tdata;

  tdata.org  = hitP + (1e-6f)*dir;
  tdata.dir  = dir;
  tdata.vptr = &self;
  tdata.eta  = self.eta0;

  // Traverse volume.
  traverse<Mode>(self, prd);
}
```

Section 36.2. The `traverse` helper function shown in Listing 36-10 implements the discretized, first-order differential equations characterizing the propagation of light through an optically inhomogeneous medium—that is, through our volume object.

Fundamentally, `traverse` computes and updates `TraversalData` values while sample points along the ray path are within the volume bounds. For each such point, refractive index gradient values are trilinearly interpolated before stepping to the next sample along the path. For reconstruction sampling, we store sample points corresponding to nonzero gradients, postponing rays when the *samples* buffer is full; postponed ray paths are resumed in the next pass of our algorithm.

Listing 36-10. *`traverse` helper function.*

```
template<Render::Mode Mode>
inline __device__
void traverse(const VolumeGeomData& self, PerRayData& prd) {
  const vec3i& dims = self.dims;
  const float  ds   = 2.f*self.step;

  // Begin volume traversal event.
  TraversalData& tdata = prd.tdata;
  tdata.event = Interaction::Event::Traverse;

  vec3f& org = tdata.org;
  vec3f& dir = tdata.dir;

  while (self.bounds.contains(org)) {
    // Fetch gradient.
    const vec3i cell   = getCell(self, org);
    const vec3f weight = org - vec3f(cell);
    const vec3f grad   = fetchGradient(self.gradient, cell, dims, weight);

    // Store sample (if necessary).
    if (Mode == Render::Mode::Sample) {
      if (length(grad) > 0.f) {
        if (!storeSample(prd))
          return;
      }
    }

    // Step to next sample.
    const vec3f porg = org;

    float& eta      = tdata.eta;
    vec3f& atten    = tdata.atten;
    int&   ndata    = tdata.ndata;
    float& distance = tdata.distance;

    org += (ds/eta)*dir;
    dir += ds*grad;
    eta += dot(grad, org - porg);

    const float len = length(org - porg);
    distance += len;

    if (self.ior_mask == nullptr || getMaskValue(self, cell)) {
      atten *= attenuate(self.absorb, len);
      ++ndata;
    }
  }

  // End volume traversal event.
  tdata.event  = Interaction::Event::Exit;

  prd.out.status = Path::Status::Bounced;
  prd.out.org    = org;
  prd.out.dir    = normalize(dir);
  prd.out.atten  = self.albedo*tdata.atten;
}
```

Ray paths are also attenuated (and samples counted, to support color-mapping in direct visualization), either always or according to an optional data mask. Once the ray path exits the volumes bounds, `TraversalData` values are updated accordingly, and the per-ray traversal outputs are stored to support shading operations in the outer path-tracing loop.

STORESAMPLE HELPER FUNCTION

As described in Section 36.3.1, the ability to postpone a ray mid-flight is critical to our multi-pass traversal algorithm. The `storeSample` helper function, shown in Listing 36-11, implements this behavior.

Listing 36-11. *`storeSample` helper function.*

```
inline __device__
bool storeSample(PerRayData& prd) {
  const int& MaxSampleDepth = optixLaunchParams.msd;

  const int& pidx     = prd.tdata.pidx;
        int& nsamples = optixLaunchParams.nsamplesBuffer[pidx];
  if (nsamples >= MaxSampleDepth) {
    prd.out.status = Path::Status::Postponed;
    optixLaunchParams.doneBuffer[pidx] = 0;

    return false;
  }

  // Store sample point.
  const vec2i&      fbSize = optixLaunchParams.fbSize;
  const int         sidx   = nsamples*fbSize.y*fbSize.x + pidx;
        SampleData& sample = optixLaunchParams.sdataBuffer[sidx];

  sample = SampleData(prd.tdata);
  ++nsamples;

  return true;
}
```

Recall that, for each pass and for each ray path, at most $N_S \leq$ `MaxSampleCount` samples are stored, where `MaxSampleCount` is the user-controlled maximum sample count. For a path with open entries in the *samples* buffer, the sample point is stored, the corresponding sample count is incremented, and the function returns `true` to indicate a successful operation.

However, for any path acquiring $N_S >$ `MaxSampleCount` samples in the current pass, the path must be postponed. Here, the path traversal status is set to *postponed*, the *done* flag is set to *incomplete*, and the function returns `false` to indicate that the operation failed.

In either case, control returns to the caller and processing continues accordingly.

MISS PROGRAM

If no intersection is found during ray traversal, the *miss program* executes; in our context, the miss program sets the path traversal status and updates the corresponding `TraversalData`, as required. These operations are shown in Listing 36-12.

Listing 36-12. *OptiX 7 miss program.*

```
OPTIX_MISS_PROGRAM(miss)() {
  PerRayData& prd = owl::getPRD<PerRayData>();
  prd.out.status  = Path::Status::Missed;

  // Store path endpoint (if necessary).
  if (optixLaunchParams.rmode == Render::Mode::Sample) {
    const vec3f org = optixGetWorldRayOrigin();
          vec3f dir = optixGetWorldRayDirection();

    dir = normalize(dir);

    prd.tdata.org   = org + dir;
    prd.tdata.dir   = dir;
    prd.tdata.event = Interaction::Event::Miss;

    storeSample(prd);
  }
}
```

OTHER ELEMENTS

The example application demonstrating curved ray traversal leverages several other processing elements—including the driver program and supporting infrastructure, test scripts for results validation, utility programs, and so forth. We omit discussion of these elements and instead refer the interested reader to the application source code for more information.

Using these components, then, our multi-pass curved ray traversal implementation provides a flexible, extensible, and efficient application framework supporting both gradient field tomography and interactive rendering.

36.4 CONCLUSIONS

We have presented a multi-pass implementation of curved ray traversal through volumes of spatially varying refractive indices using OWL, the OptiX 7 Wrapper Library.

Curved ray traversal is governed by continuous refraction for optically inhomogeneous media, which models the propagation of light in such media according to the ray equation of geometric optics. The ray equation naturally incorporates straight and curved ray paths to simulate phenomena like reflection and refraction. We demonstrate the utility of our curved ray traversal implementation using two applications: sampling for reconstruction and interactive rendering of objects with varying refractive indices.

Our multi-pass variant is motivated by problems in gradient field tomography, wherein sampling for reconstruction requires computing, storing, and later retrieving sample points along each ray. We leverage multiple sampling passes to avoid (potentially expensive) dynamic memory allocation on the device, as well as (likely wasteful and potentially insufficient) overprovisioning of device-side memory resources. The data structures, ray tracing programs, and helper functions implementing our multi-pass algorithm enable not only simulation, reconstruction, and visualization of unknown compressible flows in gradient field tomography, but also direct visualization, including interactive rendering of refractive objects.

The full source code of our application framework, distributed under a permissive open source license, enables readers to explore, modify, or enhance our curved ray traversal implementation. This foundational, flexible, and efficient implementation permits several possible extensions to enhance both visual fidelity and performance, including more sophisticated volume proxy geometries, hand-optimized *interactive-rendering-only* code paths, more sophisticated memory-management schemes, and advanced rendering features such as additional materials models and support for multiple volume geometries.

REFERENCES

[1] Atcheson, B., Ihrke, I., Heidrich, W., Tevs, A., Bradley, D., Magnor, M., and Seidel, H.-P. Time-resolved 3D capture of non-stationary gas flows. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 27(5):132:1–132:9, 2008. DOI: 10.1145/1457515.1409085.

[2] Bell, N. and Hoberock, J. Thrust: A productivity-oriented library for CUDA. In W.-M. Hwu, editor, *GPU Computing Gems*, pages 359–371. Morgan Kaufmann, 2012.

[3] Born, M. and Wolf, E. *Principles of Optics*. Cambridge University Press, 2019.

[4] Dagum, L. and Menon, R. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. DOI: 10.1109/99.660313.

[5] Gribble, C. Curved ray traversal—Source code. http://www.rtvtk.org/~cgribble/research/crt-rtg2/. Last accessed April 5, 2021.

[6] Gribble, C., Eijkhout, V., and Navratil, P. Implementing a prototype system for 3D reconstruction of compressible flow. In *Practice and Experience in Advanced Research Computing*, pages 198–206, 2020.

[7] Guennebaud, G. Eigen: A C++ linear algebra library. Presentaion at First Plafrim Scientific Day, Bordeaux, France, May 31, 2011. Slides available at http://downloads.tuxfamily.org/eigen/eigen_plafrim_may_2011.pdf.

[8] Gutierrez, D., J.Seron, F., Munoz, A., and Anson, O. Simulation of atmospheric phenomena. *Computers & Graphics*, 30(6):994–1010, 2006. DOI: 10.1016/j.cag.2006.05.002.

[9] E. Haines and T. Akenine-Möller, editors. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress, 2019.

[10] Ihrke, I. and Magnor, M. Image-based tomographic reconstruction of flames. In *Proceedings of Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, pages 365–373, 2004.

[11] Ihrke, I., Ziegler, G., Tevs, A., Theobalt, C., Magnor, M., and Seidel, H.-P. Eikonal rendering: Efficient light transport in refractive objects. *ACM Transactions on Graphics*, 26(3):59:1–59:10, 2007. DOI: 10.1145/1276377.1276451.

[12] Jacob, B. and Guennebaud, G. Eigen. https://eigen.tuxfamily.org/index.php?title=Main_Page. Last accessed April 5, 2021.

[13] Kajiya, J. T. The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, pages 143–150, 1986.

[14] Morley, K. How to get started with OptiX 7. https://developer.nvidia.com/blog/how-to-get-started-with-optix-7/, November 20, 2019. Last accessed April 5, 2021.

[15] NVIDIA. NVIDIA OptiX 7.2—Programming guide. https://raytracing-docs.nvidia.com/optix7/guide/index.html. Last accessed April 5, 2021.

[16] NVIDIA. NVIDIA RTX platform. http://developer.nvidia.com/rtx. Last accessed April 5, 2021.

[17] NVIDIA. Thrust. https://developer.nvidia.com/thrust. Last accessed April 5, 2021.

[18] OpenMP Architecture Review Board. OpenMP application program interface. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015. Last accessed April 5, 2021.

[19] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 3rd edition, 2017.

[20] Raffel, M. Background-oriented schlieren (BOS) techniques. *Experiments in Fluids*, 56:60:1–60:17, 2015. DOI: 10.1007/s00348-015-1927-5.

[21] Shirley, P. *Ray Tracing in One Weekend*. 2016. https://raytracing.github.io. Last accessed April 5, 2021.

[22] Stam, J. and Langénou, E. Ray tracing in non-constant media. In *Eurographics Workshop on Rendering Techniques*, pages 225–234, 1996.

[23] Sun, X., Zhou, K., Stollnitz, E., Shi, J., and Guo, B. Interactive relighting of dynamic refractive objects. *ACM Transactions on Graphics*, 27:35:1–35:9, 2008. DOI: 10.1145/1399504.1360634.

[24] Wald, I. OWL—The OptiX 7 wrapper library. https://owl-project.github.io. Last accessed April 5, 2021.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 37

RAY-TRACING SMALL VOXEL SCENES

Dylan Lacewell
NVIDIA

ABSTRACT

We build acceleration structures for voxel/brick scenes using NVIDIA OptiX and render them in a non-photorealistic style with outlines and one bounce of indirect lighting; this method runs at interactive rates on recent RTX hardware and is implemented entirely in OptiX shaders with no post-processing.

37.1 INTRODUCTION

In this chapter we present a method for rendering *voxel art*: minimalist 3D scenes constructed entirely of uniformly sized, axis-aligned bricks with an



Figure 37-1. *A scene from Mini Mike's Metro Minis [6] ray-traced with NVIDIA OptiX 7 and the OptiX Wrapper Library. The outline effect is also ray-traced in the same kernel.*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_37

8-bit color palette evocative of early video games. These scenes are built using MagicaVoxel [3], or converted from *Minecraft* files or meshes, as a starting point. Artists typically do final renders with path tracing, either using the built-in renderer in MagicaVoxel (which we assume intersects bricks directly) or by exporting a triangle mesh to an external renderer.

There is a long history in graphics of tracing voxels as volume elements, but not necessarily as bricks. The closest work to ours is the paper from Majercik et al. [7], which sped up primary visibility tests for dynamic bricks in a hybrid renderer, but did not focus on acceleration structures or shading as we do.

Our descriptions are geared toward the NVIDIA OptiX API and the OptiX Wrapper Library (OWL) [10], but readers familiar with DirectX Raytracing or Vulkan Ray Tracing should be fine.

37.2 ASSETS

The first obstacle is finding nice (and free) input scenes, preferably in the MagicaVoxel file format, *VOX*, which is relatively simple compared to *Minecraft* files. Though there are many renders of amazing MagicaVoxel scenes online, there are few downloadable VOX files. One notable exception is the *Mini Mike's Metro Minis* collection [6], more than 400 scenes licensed under Creative Commons and available in a public repository. These scenes show characters in an urban setting doing various mundane (and sometimes disturbing) things.

37.3 GEOMETRY AND ACCELERATION STRUCTURES

Each VOX scene (a single file) is a collection of instanced models with transforms and a small global palette of 8-bit RGB values. Models are axis-aligned grids of 1-byte indices into the color palette. Some models have extra material parameters such as roughness values, which we currently ignore.

We read VOX files using an open source library [5] and then extract the nonempty voxels into a linear array of bricks per model, to be used as primitive inputs for OptiX—each brick is an (x_i, y_i, z_i, c_i) index tuple. We could optionally remove hidden bricks here, but do not do so by default, to allow for clipping planes and other dynamic changes (Figure 37-2).

There are several ways to build an OptiX scene from these primitives, as shown in Figure 37-3, and the trade-offs between these was not initially clear.

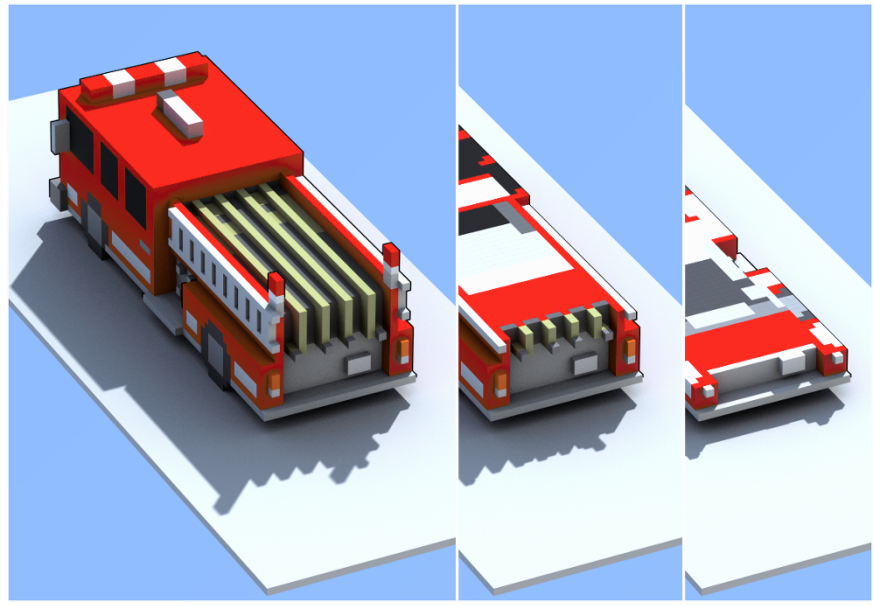


Figure 37-2. *We preserve all bricks in the input model, which allows for clipping planes.*

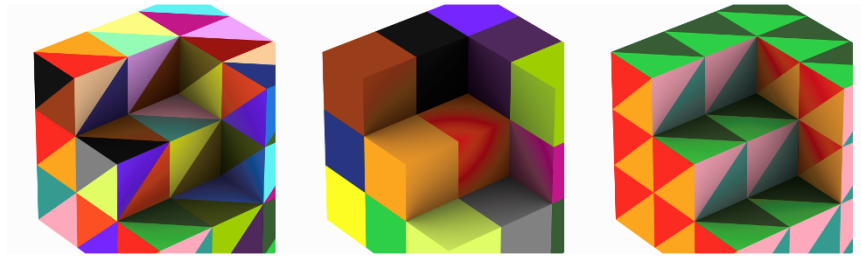


Figure 37-3. *Different ways to build a scene for a VOX model. Left: flat triangle mesh. Center: bricks as custom primitives. Right: instanced triangle brick.*

37.3.1 FLAT TRIANGLE MESH

A traditional approach is to triangulate each brick and concatenate all triangles into a single mesh, then build a Geometry Acceleration Structure (GAS) over the mesh. This takes full advantage of ray tracing hardware for both acceleration traversal and ray/triangle intersection. However, bricks are volumetric and brick models have a lot of surface area per volume; representing each brick as 12 triangles uses significant device memory, with even more needed for modeled bricks with bevels or studs.

Static meshes could be more aggressively optimized, at least with simple six-sided bricks, e.g., by merging coplanar faces from adjacent bricks [1]. However, these changes are destructive and would rule out any edits to bricks, or changes in transparency later. In our implementation we share vertices between bricks where possible, but do not otherwise optimize the mesh.

37.3.2 CUSTOM INTERSECTION PROGRAM

A brick can also be represented as a *custom* primitive in OptiX with its own intersection and bounds programs. This moves ray/brick intersection into software, although rays still traverse the acceleration structure in hardware. For this option we implemented the fast ray/brick intersection algorithm from Majercik et al. [7] in CUDA/OptiX with some simplifying assumptions: each brick is axis aligned and has unit dimensions in object space, and rays are assumed to start outside bricks (a ray that starts slightly inside a brick due to biasing will miss the brick).

The upside is that custom primitives have far less memory usage than a flat mesh. The intersection program only needs access to the array of 4-byte (x_i, y_i, z_i, c_i) voxel indices, the color palette, and the model dimensions in world space.

Voxels can also be grouped into small blocks, e.g., $8 \times 8 \times 8$ voxels as a custom primitive, and traversed in software using a grid digital differential analyzer (DDA) algorithm [9, 4]. Blocks use less memory than single bricks because they have fewer (x_i, y_i, z_i) indices and fewer leaf nodes in the RTX acceleration structure, but on the other hand, this approach does not utilize RTX hardware as much at the leaf level.

37.3.3 INSTANCED TRIANGLE BRICK

A third option is to represent a single brick as triangles in a small GAS and to instance the brick over the model with an instance acceleration structure (IAS) rather than a flat mesh. OptiX exposes a *visibility mask* per instance (but not per primitive), which we found useful for some shading effects described later. The memory cost of 12 floats per affine transform is in between the memory usage of the flat mesh and custom primitive options. Instancing is an efficient way to support more heavyweight bricks like the beveled bricks shown in Figure 37-4.

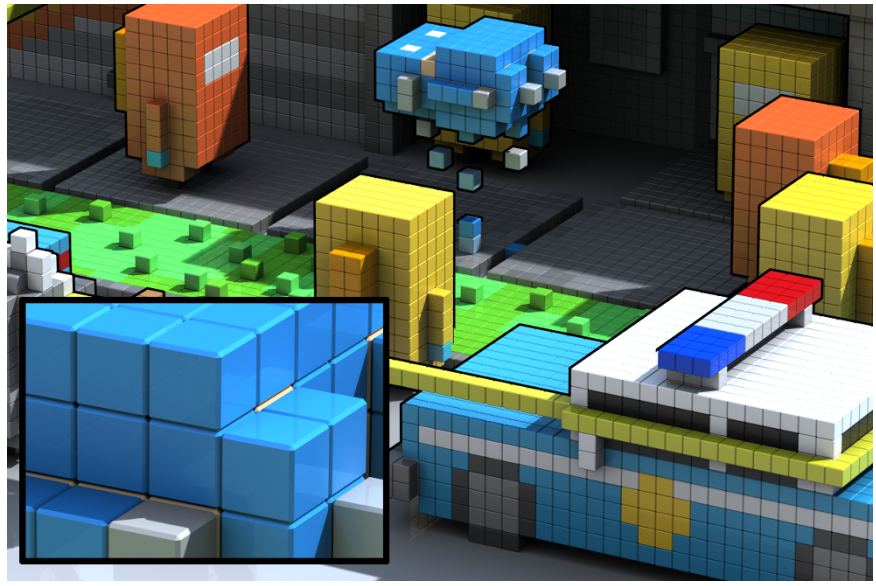


Figure 37-4. *Beveled bricks, cheap to instance but expensive to triangulate. The bricks shown here have 44 triangles each.*

37.4 SHADING

Our basic path tracing loop is similar to other OptiX/OWL samples, and we won't dwell on it (iterative path tracer in the ray generation shader, shadow rays in the closest-hit shader). We focus on the more unusual feature of our sample renderer, the cartoon-style outlines that are shown in most figures.

Previous methods for drawing outlines in a ray tracer operate mostly in image space, either detecting depth edges with extra stencil rays [2] or analyzing arbitrary output variables (AOVs) at nearby samples [8]. These image-space methods provide artistic control over line properties, although they do not easily handle defocus effects like motion blur or depth of field.

Alternatively, some real-time games use a multi-pass method for outlines, sometimes called the *inverted hull method*. In a first pass geometry is drawn normally, then in a second pass all triangles are inflated slightly along their normals and then rendered with frontface culling and a constant color; this creates an outline because the *hull* geometry as constructed only appears where there is a change in depth between pixels. This method does not provide as much artistic control but is arguably easier to implement in a rasterizer, and is compatible with motion blur and depth of field.



Figure 37-5. *Constructing outlines. Top left: original bricks. Top right: scaled and inverted hull geometry (with shading for visualization). Bottom left: outlines with a depth bias of 0. Bottom right: outlines with a depth bias of five bricks.*

As an experiment, we adapted this multi-pass rasterization method to ray tracing as follows; also see Figure 37-5:

1. Build a second acceleration structure containing the hull geometry, with each brick scaled up slightly about its center (we scale by 1.2 in all figures). This does not necessarily need to be a separate acceleration structure in practice, as long as it can be distinguished from the original geometry during traversal.

2. At each subpixel sample, trace a primary camera ray into the regular scene to find the closest hit, and return its distance in per-ray data.
3. After the primary ray, immediately shoot a special shadow ray using the same origin and direction, but traced against the hull acceleration structure and using the distance from the primary ray as a t-max value (a tiny depth buffer, essentially). Set the OptiX ray flags to cull frontfaces. This step returns a visibility value V of 0 if the ray hits the hull geometry and 1 otherwise.
4. Independently, finish the path for the primary ray, tracing any bounces and shadow rays and returning a radiance R.
5. Return $V \times R$ as the final path radiance—black for an outline.

It is tempting to check the value of V before finishing the bounces in step 4, but this introduces a dependency and is slightly slower with simple Lambertian shading (but might still be a good idea for more expensive shading).

The amount of extra memory used by the hull geometry depends on how primitives are represented in the original scene. For the flat triangle mesh approach, it is hard to avoid a second copy of the entire scene in memory. Note that we cannot simply insert a single new scaled instance of the scene; each brick needs to be scaled locally.

For custom primitives with one brick per primitive, we can represent the hull geometry in the same acceleration structure with no increase in memory if we scale the user-defined boxes slightly (a displacement bound, basically). We assign a different intersection program to each ray type: radiance rays intersect a unit box, and shadow-hull rays intersect a scaled box and don't need to compute normals.

The effect is harder to implement if custom primitives contain multiple bricks in grids, because of the scaling; bricks would need to be inserted in neighbor cells, and this would complicate the DDA traversal. We did not implement this.

For the instanced bricks approach, we need a second copy of each transform for the locally scaled brick. We store both the original and new transforms in the same acceleration structure, using OptiX visibility flags to hide the hull geometry from radiance rays, and vice versa. This is approximately a double memory increase however—not cheap.

Here are a few more details on traversal: As with the original rasterization-based algorithm, a depth bias parameter for the hull shadow

ray can be used to control whether outlines appear around every brick or at a larger scale. Also note that we avoid issues with sharp edges, which are typically a problem in the original algorithm, because we scale bricks about their centroids rather than inflating vertices along normals.

As an object-space method, the resulting outlines are not necessarily uniform in screen space, depending on the field of view. This may or may not be desirable. However, brick models are often rendered using a camera with a small field of view for aesthetic reasons anyway.

37.5 PERFORMANCE TESTS

Though a deep analysis of performance is out of the scope of this chapter, we measured GPU memory and kernel time for two scenes: first, the small scene from Figure 37-1, which has 63,481 bricks; then, a larger merged scene shown in Figure 37-6, which has 4.94 million bricks. The memory reported is for buffers and acceleration structures, and we did not cull hidden bricks.

We used NVIDIA Nsight Systems to measure average kernel times. Each kernel launch traces one path per pixel, and a path has at most four rays (primary, shadow, bounce, and shadow), plus one extra ray when outlines are enabled. As a side note, we found that accumulating samples was always

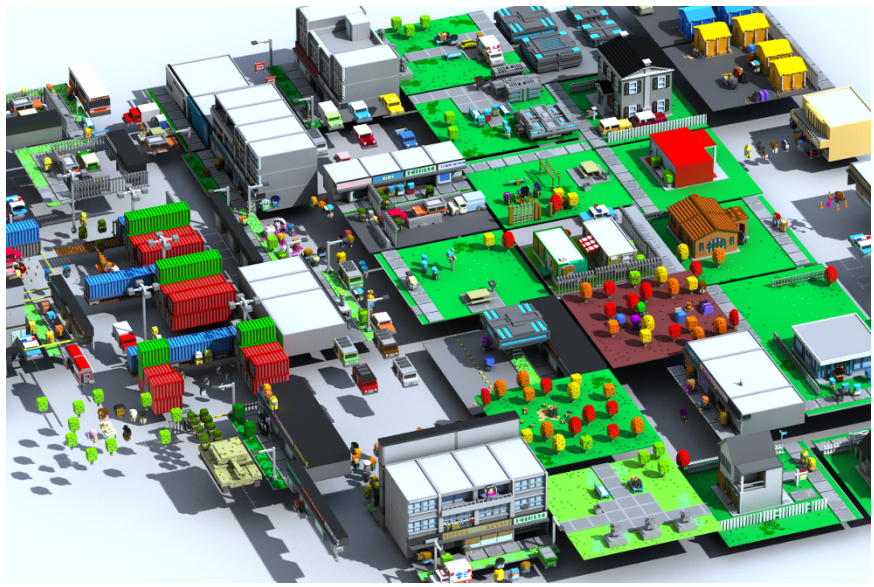


Figure 37-6. *Multiple scenes merged together for a performance test (4.94 million bricks).*

	Device Mem (MB)		Kernel Time (ms)	
Small Scene		w/outlines		w/outlines
Triangle mesh	50.19	--	1.48	--
Custom-1	3.37	3.37	1.54	2.53
Custom-8	0.22	--	2.90	--
Instanced brick	25.64	51.21	1.38	1.84
Merged Scene				
Triangle mesh	3903.56	--	2.75	--
Custom-1	261.71	261.71	2.45	3.63
Custom-8	12.76	--	4.29	--
Instanced brick	1244.31	2443.67	2.34	3.15

Table 37-1. *Scene memory (geometry and acceleration structures) and average kernel time over 4800 launches, for two scenes (*1200 $\times$ 800 *images, NVIDIA RTX 3060 Ti).*

	Primary	Bounce	Shadow	Outline	Total
Small Scene	960,000	957,978	811,733	960,000	3,689,711
Merged Scene	960,000	960,000	1,051,624	960,000	3,931,624

Table 37-2. *Ray counts per launch for the two scenes in Table 37-1.*

faster using separate launches versus doing two, four, etc. samples per pixel in each launch.

Results are shown in Table 37-1 for the different acceleration layouts discussed earlier, and the related ray counts are shown in Table 37-2. For custom primitives we tried both a single brick per prim (Custom-1) and an $8 \times 8 \times 8$ grid per prim (Custom-8). Please note however that DDA traversal for the $8 \times 8 \times 8$ grid is not a trivial intersection shader, and there is possibly more room for optimization there. We also show results with the outline effect for the Custom-1 and Instanced approaches, where outlines were easiest to implement.

Custom primitives use far less memory than other options, especially when grouped into $8 \times 8 \times 8$ blocks. Kernel times are surprisingly fast for Custom-1, compared to full hardware acceleration; bricks might be an especially good case for custom primitives because they fill their axis-aligned bounding boxes perfectly, but a difficult case for triangles without optimizing for static bricks.

The outline effect increases the total kernel time by about 60% for custom primitives because of the padded bounding boxes, but only about 30% for

instanced bricks, which roughly matches the increase in total ray count of 35% due to the extra outline rays.

37.6 DISCUSSION

From the performance measurements above, we conclude that very large voxel/brick scenes are probably best represented as custom primitives; however, for smaller scenes, instancing is faster and gives more flexibility in how each brick is modeled. To further reduce memory, small blocks of bricks could be instanced instead of single bricks. For example, representing every $2 \times 2 \times 1$ block as an instance should reduce memory, as there are only five unique patterns up to rotation, and the rotation can be baked into the instance transform.

It would also be interesting to look more at dynamic changes to the scene, e.g., adding or removing bricks on the fly and refitting the acceleration structures as an editing package would need to do.

Complete sample code for this chapter is available in the OWL repository. We have only scratched the surface of brick rendering, and we encourage readers to download the set of VOX models shown here and to experiment with other types of shading and lighting.

ACKNOWLEDGMENTS

We would like to thank Keith Morley and David Hart for many discussions about ray tracing bricks. And thanks to Mike Judge, for making your art available!

REFERENCES

[1] Barrett, S. stb. https://github.com/nothings/stb/blob/master/stb_voxel_render.h, 2021. Accessed February 2021.

[2] Choudhury, A. N. M. I. and Parker, S. G. Ray tracing NPR-style feature lines. In *NPAR '09: Proceedings of the 7th International Symposium on Non-photorealistic Animation and Rendering*, pages 5–14, 2009.

[3] ephtracy. MagicaVoxel. https://ephtracy.github.io/, 2021. Accessed February 2021.

[4] gltracy. MagicaVoxel. *Shadertoy*, https://www.shadertoy.com/view/MdBGRm, 2013. Accessed April 2021.

[5] jpaver. Open game tools. https://github.com/jpaver/opengametools, 2021. Accessed February 26, 2021.

[6] Judge, M. Mini Mike's Metro Minis. https://github.com/mikelovesrobots/mmmm, 2021. Accessed February 2021.

[7] Majercik, A., Crassin, C., Shirley, P., and McGuire, M. A ray-box intersection algorithm and efficient dynamic voxel rendering. *Journal of Computer Graphics Techniques*, 7(3):66–81, 2018. http://jcgt.org/published/0007/03/04/.

[8] Ogaki, S. and Georgiev, I. Production ray tracing of feature lines. In *SIGGRAPH Asia 2018 Technical Briefs*, 15:1–15:4, 2018. DOI: 10.1145/3283254.3283273.

[9] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 3rd edition, 2016.

[10] Wald, I. OWL: A node graph "wrapper" library for OptiX 7. https://github.com/owl-project/owl, 2021. Accessed March 2021.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





PART VI
PERFORMANCE

PART VI

PERFORMANCE

Though the film industry may hope to render a frame in sixty seconds, in video games and other interactive applications, we would hope to render sixty frames or more in a single second, leaving scant milliseconds for each image to be produced. This part contains chapters that focus on a variety of topics that explore the optimization space of ray tracing performance, proposing both algorithmic and heuristic improvements to aid in the relentless search for both faster runtime speed as well as smaller memory utilization.

Chapter 38, *CPU Performance in DXR*, discusses several practical methods for improving the performance of DirectX Raytracing—based applications in order to address memory and compute requirements. These include attention to state object collections, incremental compilation, careful shader table structure, and amortized acceleration structure rebuilds.

Chapter 39, *Inverse Transform Sampling Using Ray Tracing Hardware*, describes how ray tracing hardware can be used to perform inverse transform sampling, mapping an image texture to geometry for hardware-accelerated ray testing. This chapter uses emissive textures such as environment maps as a motivating example.

Chapter 40, *Accelerating Boolean Visibility Operations using RTX Visibility Masks*, details how to improve on the performance of rendering clipped geometry for inspecting detailed scenes. The authors show how to transform costly evaluations of any-hit shaders into efficient, hardware-accelerated visibility masks representing a disjunction over Boolean-valued visibility functions.

Chapter 41, *Practical Spatial Hash Map Updates*, extends the previous spatial hashing technique and generalizes the mechanism to support the update of arbitrary data types. This is accomplished via the use of novel change lists, which are later resolved and committed. Practical applications in ambient occlusion and environment lighting are discussed.

Chapter 42, *Efficient Spectral Rendering on the GPU for Predictive Rendering*, makes the case for spectral rendering in contrast to traditional three-color rendering. Given this base assumption, the chapter then explores approaches

to optimize intermediate buffers, wavelength selection, and spectral asset management. Though real-time fully converged results are left for future work, the authors show denoised low-sample results up to around 30 images per second.

Chapter 43, *Efficient Unbiased Volume Path Tracing on the GPU*, presents several optimizations to improve the performance of volumetric path tracing. These approaches improve not only runtime performance but also memory consumption, reducing it by a factor of 6.5. The use of a multi-level digital differential analyzer and indirection texture improve execution time by two to three times.

Chapter 44, *Path Tracing RBF Particle Volumes*, discusses the use of a radial basis function (RBF) model for volume rendering. An example implementation is demonstrated using OSPRay and the Open Volume Kernel Library, and the chapter includes pseudocode of the open source implementation. The authors evaluate performance on a 51 million particle volume on a 28 core Intel Xeon processor.

Chapter 45, *Fast Volumetric Gradient Shading Approximations for Scientific Ray Tracing*, details a volume rendering optimization by evaluating illumination at a single point along the visibility ray. This one simple trick increases performance of the ParaView rendering by 5–50×.

We hope this part will provide valuable insight for achieving previously unthinkable levels of performance through ray tracing.

Josef Spjut and Michael Vance

CHAPTER 38

CPU PERFORMANCE IN DXR

Peter Morley
NVIDIA

ABSTRACT

DirectX Raytracing (DXR) performance guides have mainly focused on accelerating ray tracing on the graphics processing unit (GPU). This chapter will focus on avoiding stalls and bottlenecks caused by DXR on the central processing unit (CPU) side.

38.1 INTRODUCTION

Most game engines must deal with streaming assets in and out of memory as the player moves throughout the built environment. Typically, the game world space is partitioned into discrete tiles or volumes, with only the elements close to the player being resident in memory, and optionally placeholders or reduced level of detail (LOD) geometry standing in for more distant items.

Implementing DXR ray tracing into a renderer adds several significant memory and compute overheads. This chapter presents techniques to optimize the management of acceleration structures and shader tables.

Methods to reduce DXR CPU overhead include the following:

- Use generic hit group shaders and use precompiled collections for a state object.
- Use incremental state object compilation.
- Reduce shader table complexity for the local root signature.
- Limit acceleration structure (AS) builds and refits.

38.2 THE RAY TRACING PIPELINE STATE OBJECT

The ray tracing pipeline state object (RTPSO) contains a network of shaders defining how various materials will be processed in a ray traced scene. The following techniques that will help reduce the amount of CPU overhead

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_38

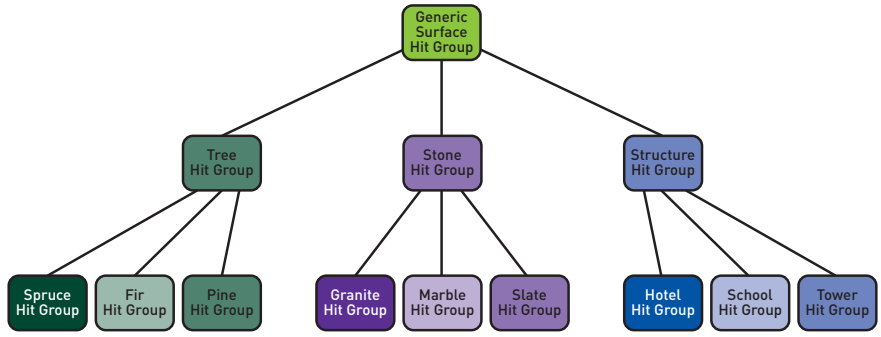


Figure 38-1. *The nine hit group shaders can be consolidated into a single generic surface hit group by unifying how common materials are shaded. Engines will separate hit groups into categories such as transparent, decals, generic, and subsurface materials, to name a few.*

associated with the RTPSO include reducing hit group shaders, as well as incremental changes to the RTPSO and state object collection multi-threaded compilation.

One way to mitigate heavy shader permutations is to break down the shader tables into generic hit group shaders that handle a collection of geometry types in order to make the number of hit groups in the RTPSO manageable. Figure 38-1 shows a state object configuration in which unique hit group shaders can be unified by functionality. Having a predefined set of hit group shaders in a state object is important because it not only reduces shader execution divergence but also reduces CPU delay from recompiling the state object every time new hit group shaders are needed.

38.2.1 INCREMENTAL STATE OBJECT MODIFICATIONS

Ray tracing pipeline state object compilation can be computationally expensive due to the driver compiling multiple shaders. `AddToStateObject` was introduced to prevent recompiling the entire RTPSO and only requires compilation of new hit group shaders as they are added. `AddToStateObject` is a very lightweight operation because the Direct3D 12 runtime does not need to validate the entire state object and the driver only needs to perform a trivial linking step after the new shaders are compiled. If complex state objects can't be avoided, then it is best practice to compile the most common hit group shaders into a state object once and incrementally add new hit group shaders with `AddToStateObject` during gameplay. The full documentation for `AddToStateObject` is available [2].

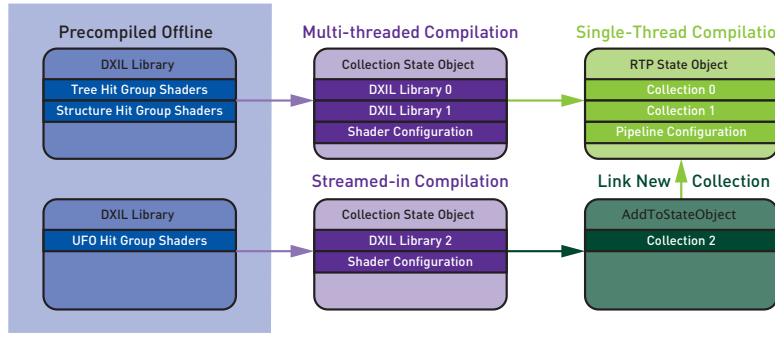


Figure 38-2. *Precompiled DXIL binaries are compiled into state object collections at startup then finally compiled into a ray tracing pipeline state object.*

38.2.2 STATE OBJECT COLLECTIONS

Another useful strategy to reduce the amount of compilation time for ray tracing pipeline state objects is to use state object collections. Collections containing ray tracing shader DirectX Intermediate Language (DXIL) libraries can be compiled separately and in parallel on the CPU. (See Figure 38-2.) RTPSOs can then reference these precompiled collections and in turn reduce the amount of time it takes to prepare a ray tracing pipeline. Using multiple CPU cores to compile collections can give a significant speedup compared to compiling the RTPSO with raw DXIL libraries on a single CPU core. RTPSO compilation should be lightweight due to multi-threaded collection compilations. New hit groups can be streamed in by asynchronously compiling new collections and using `AddToStateObject` to perform a trivial linking step, assuming the collection state objects were compiled by the driver.

38.3 THE SHADER TABLE

Generating complex shader tables is often a costly process on the CPU. A shader table contains an array of shader records that define the resource bindings of each top-level instance's local root signature in the top-level acceleration structure (TLAS).

38.3.1 BUILDING THE LOCAL ROOT SIGNATURE ON THE GPU

A local root signature (LRS) can use up to 4 KB, according to the DXR functional spec [3], in memory for descriptors and other resource data. One special advantage of the LRS is that the underlying memory can be allocated

in GPU memory for low latency during traversal shading. If the LRS uses constant buffers or root constant buffers, it can be very costly to constantly update those memory resources on the CPU side for a large amount of shader records. Instead, shader tables can be generated using a compute shader to write out shader table resource data by reading CPU system memory buffers containing the scene's resource information. Another advantage of building the shader table on the GPU and using GPU visible only memory is that this will avoid Peripheral Component Interconnect Express (PCIe) traffic reads (GPU to CPU memory) when accessing constant buffers and all the other resource descriptors.

- LRS pros:
  - Root signature memory size is 4 KB.
  - CPU overhead to update shader records.
- LRS Cons:
  - Root signature memory is configurable.

38.3.2 GLOBAL ROOT SIGNATURE

One limitation of the global root signature (GRS) is that if the resources required for state object shading exceed the memory capacity of a GRS, then resource management will require special attention. Another limitation is that a GRS is limited to 64 DWORDS (256 B) and only one can be bound. More information on limitations are defined by White and Satran [6].

For example, if root constants are needed to manage resource indexing, then the root constants would be accumulated into a single structured buffer and accessed with a shader resource view (SRV).

- GRS pros:
  - Unified root signature management.
  - No shader table management if LRS isn't used.
- GRS cons:
  - Root signature memory size is 256 B.
  - Root signature memory is required to be in system memory.
  - Requires alternative management of large root data sets.

38.3.3 GRS VERSUS LRS

A GRS is limited to 256 B and only one can be bound, while a LRS allows up to 4 KB and can be instanced. The LRS gives each instance in the TLAS access to 4 KB of resource memory rather than being limited to 256 B with a GRS. Using a LRS is a convenient way to get around some of the limitations of a GRS, but at the cost of CPU performance and more memory consumption.

38.3.4 SHARING RESOURCES WITH THE RASTERIZER

If the shader table construction can't be performed on the GPU, then techniques must be used to reduce CPU overhead of managing the resource views (constant buffer view, shader resource view, or unordered access view). This can involve extracting resources required for ray tracing from the already established rasterization engine's resources. Resource view management for the LRS typically involves heavy use of `CopyDescriptors` or `CopyDescriptorsSimple` to pull from a repository of views stored in a single large descriptor heap. Typically, these copies are required for hybrid (rasterization and ray tracing) renderers. The same resource sharing applies when the rasterizer consumes new vertex and index buffers needed to be rendered, which then must be passed along to the AS builder to include it during ray traversal.

38.3.5 BINDLESS RESOURCE ARRAYS

The preferred solution, if possible, is to avoid building the shader table altogether by dynamically indexing into bindless resources. Bindless resources are implemented using a descriptor table that contains an array of resource views. Large descriptor tables are stored in the GRS and can be accessed based on the instance and geometry index of the intersected primitive. (See also Chapter 17.)

38.4 THE ACCELERATION STRUCTURE

38.4.1 OVERVIEW

Processing millions of triangles into the ray tracing AS can become prohibitively expensive on both the GPU and the CPU. The first challenge is managing the transient geometry in the AS that enters and exits the player's rendering volume. For each frame, new bottom-level acceleration structures (BLAS) need to be built for new geometry that entered this volume, and conversely, existing acceleration structures need to be deallocated if they are

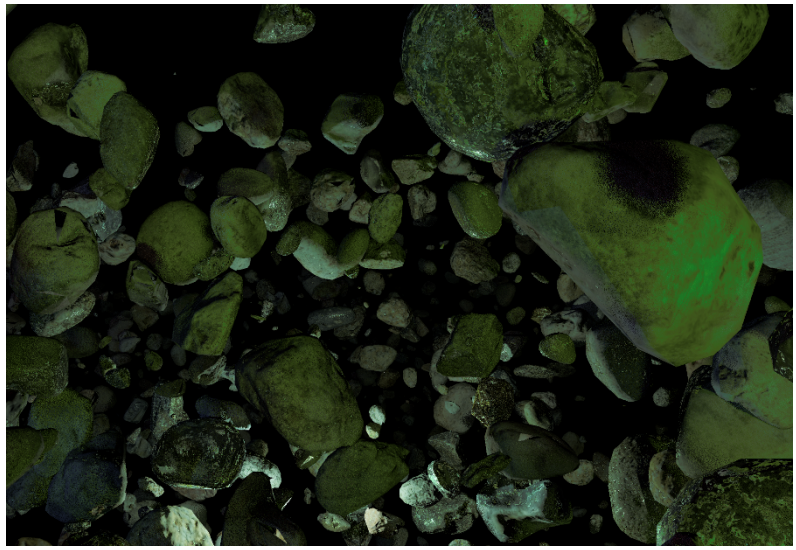


Figure 38-3. *A large collection of asteroids each stored in a unique BLAS structure. Frustum culling techniques are used to reduce the BLAS memory footprint and TLAS size.*

no longer in view. If the AS is being used for reflections or shadows, then all geometry within a certain radius of the player is required, not just the geometry in the view frustum. The movement of geometry in and out of the AS causes multiple trips to the operating system memory manager. These memory requests can introduce CPU overhead.

Figure 38-3 shows an asteroid field in which top-level asteroid instances are culled from the AS if not in the radius of inclusion. The TLAS will instance into each visible asteroid BLAS, while asteroids outside of the radius of inclusion will not be included in the TLAS and those BLAS can be deallocated.

One tool for avoiding such CPU-side memory allocation stalls involves sub-allocating the AS buffer memory. Infrequently requesting large memory pools from the operating system (OS) reduces CPU overhead because the common case would be to sub-allocate from an existing memory block. Compaction also helps in reducing the memory required for an AS by trimming the conservative memory allocation that was required for the initial build. An AS that uses compaction significantly decreases the memory footprint and, in turn, reduces CPU overhead as an added benefit. Less memory for the AS means less requests to grab sub-allocator memory blocks from the OS.

38.4.2 SHARING RESOURCES WITH THE RASTERIZER

The vertex and index buffer resources used to build the AS can hopefully be reused from the rasterization resources if the buffers are in one of the acceptable build formats. Most engines compress their vertex buffers and are forced to decompress and duplicate resources when getting ready to build the AS. The RT Cores can only interpret triangles in the form of 16- or 32-bit precision vertices. More information about RT Cores can be found in the Ampere white paper [4]. If the engine must duplicate the vertex buffers to build the AS, then be aware that deallocating those duplicated resources after building is recommended to reduce memory consumption.

38.4.3 DEFORMABLE, ANIMATED, AND STATIC AS BUILDS

There are three main categories of geometry types in the AS. These types can be described as static, animated, and deformable geometry. *Static* geometry can be defined as a triangle mesh that is uniformly transformed, such as buildings, roads, and signs. *Animated* geometry comprises a triangle mesh that has a grouping of triangles within the mesh that are transformed but adhere to the original topology, such as character animations. *Deformable* geometry is characterized by a triangle mesh in which all triangles can be transformed arbitrarily without maintaining the original topology, such as particle effects.

Static geometry has a major benefit in which it only requires a single full build. Animated geometry requires an upfront build as well but requires fast build updates every key frame, which are much less impactful on GPU performance than full builds. This requires a mesh skinning compute pass and a fast build per frame to update the AS bounding boxes for the deformed triangles. One method to reduce build processing times is to have a higher ratio of static objects compared to dynamic objects in the TLAS. Static objects only require a one-time build, whereas dynamic objects can require an update build or complete rebuild per frame. Both animated and static geometry can be compacted but require extra memory in which both the original build and the compacted build memory are resident.

Static topologies with extreme animation (running animations) or objects that change topology altogether (breakables or particles) require a full build during each of the deformable geometry's updates to maintain optimal traversal performance. AS traversal performance will degrade as triangles move farther away from their original build location and thus hurts

performance as time progresses. The trade-off is to do a full build of the deformable geometry every *N* frames and do refit builds between the *N* frames. This maintains acceptable build and traversal performance but requires tweaking for each individual engine. Putting a limit on the number of AS builds and updates per frame can also be a good way to maintain stable GPU/CPU performance with build workloads. One trade-off is that sometimes objects may appear in the AS a frame or two after they would have been traced against and show up as popping geometry. The RTX best practices blog post [5] details best practices for managing acceleration structures.

38.4.4 IMPROVING LOD PERFORMANCE

If the engine employs a LOD system, then the AS build workloads increase due to extra builds for the LODs and the management of the LODs included in the TLAS. For example, if the LOD system contains four levels of detail, then it is possible that certain assets transitioning through the LODs have the potential need to be built and compacted four separate times.

Instead, having only one LOD resident in memory prevents LOD transition popping, which is another problem set described in a stochastic LOD blog post [1]. The method described in the blog post implements smooth transitions between LODs in the acceleration structure by randomly masking out one of the LODs in the TLAS for each ray. This facilitates a gradual transition of geometry between the two LODs but at the cost of having to maintain more geometry in the AS.

One precaution with LOD selection is the potential to introduce position data bugs between rasterization and ray tracing. LOD mismatch can result in self-shadowing corruption if the acceleration structure is only using a single BLAS for each model. A technique used to mitigate self-shadowing in ray tracing is to add a bias to the ray origin. Figure 38-4 shows the self-shadowing bug when mismatching LODs for primary rays and rasterization.

The benefit to avoiding an LOD system for ray tracing prevents not only a reduction in BLAS memory but also a reduction in AS builds. The trade-off here is that geometry that is far away and highly tessellated will potentially hurt traversal performance. The TLAS is designed to prevent wasted traversal time in these highly tessellated regions, so the performance trade-off might not be as bad as expected.

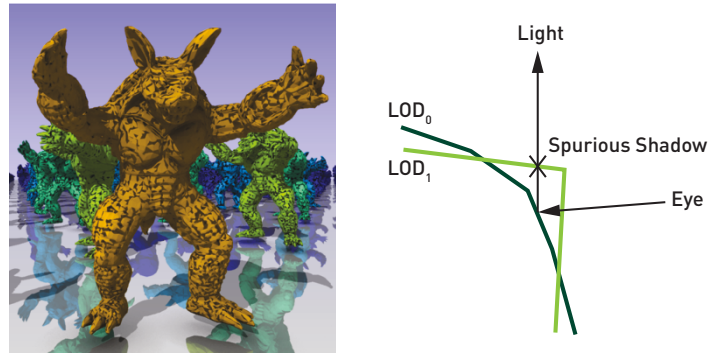


Figure 38-4. *The BLAS and rasterization vertex buffer data is different due to LOD mismatch, and the result is self-shadowing artifacts [1, Figure 3].*

38.5 CONCLUSION

The intention of this chapter is to help better understand the pros and cons associated with certain DXR design choices and how they affect CPU performance. In conclusion, the recommended approach is to simplify the RTPSO as much as possible and asynchronously compile state collections for inclusion in the RTPSO. Allocate shader tables in GPU memory and build them on the GPU to reduce CPU overhead and improve traversal shading performance. Limit the number of AS builds per frame, as to reduce memory requests to the OS and reduce the amount of AS build times on the GPU. Implementing LOD algorithms for DXR introduces a significant amount of extra memory and additional AS builds but succeeds in preventing LOD mismatch between the rasterizer and the ray tracer.

REFERENCES

[1] Lloyd, B., Klehm, O., and Stich, M. Implementing stochastic levels of detail with Microsoft DirectX Raytracing. *NVIDIA Developer Blog*, https://developer.nvidia.com/blog/implementing-stochastic-lod-with-microsoft-dxr/, June 15, 2020.

[2] Microsoft. ID3D12Device7::AddToStateObject method (d3d12.h). *Windows Developer*, https://docs.microsoft.com/en-us/windows/win32/api/d3d12/nf-d3d12-id3d12device7-addtostateobject, September 15, 2020.

[3] Microsoft. Local root signatures vs global root signatures. *DirectX Raytracing (DXR) Functional Spec*, https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html#local-root-signatures-vs-global-root-signatures, 2021.

[4] NVIDIA. Ampere GA102 GPU architecture: Second-generation RTX. White paper, https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, 2021.

[5] Sjoholm, J. Best practices: Using NVIDIA RTX ray tracing. *NVIDIA Developer Blog*, https://developer.nvidia.com/blog/best-practices-using-nvidia-rtx-ray-tracing/, August 10, 2020.

[6] White, S. and Satran, M. Root signature limits. *Windows Developer*, https://docs.microsoft.com/en-us/windows/win32/direct3d12/root-signature-limits, May 31, 2018.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 39

INVERSE TRANSFORM SAMPLING USING RAY TRACING HARDWARE

Nate Morrical[1] and Stefan Zellmann[2]
[1]NVIDIA
[2]University of Cologne

ABSTRACT

Accurate radiance estimates of high dynamic range textures require importance sampling to direct rays toward influential regions. However, traditional inverse transform sampling involves several expensive searches to locate these highly influential pixels. We propose a reformulation of inverse transform sampling that replaces these texture space searches with a single hardware-accelerated ray traversal search using cumulative probability geometry. We evaluate the performance and scalability of our approach on a set of emissive dome light textures and demonstrate significant improvements over traditional solutions.

39.1 INTRODUCTION

Whether working within real-time constraints or striving for maximum quality, ray tracing is a balancing act between improving performance and reducing variance. For real-time ray tracers, performance is of utmost importance. Variance reduction strategies must stay within a particular performance budget, even if that means choosing a cheaper, less effective noise reduction strategy over a more effective, but also expensive, method. In offline settings, variance reduction is usually a higher priority, and so some sacrifices might be made to interactivity as long as overall convergence times improve. In either case, there is a high demand for techniques that improve performance while simultaneously reducing noise.

In this chapter, we focus our attention toward improving the state of the art in large emissive texture importance sampling. Emissive textures are a great

Both authors contributed equally to this work.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_39



Figure 39-1. *A collection of emissive textures used for testing importance sampling. Many of these textures contain many low-influence texels and a small set of highly influential texels.*

way to increase the detail and overall realism of a scene, and they are frequently used to create quick, yet effective, environmental lighting. Unfortunately, if sampled naively, these emissive textures are also a common source of noise. With many emissive textures, the majority of the incoming radiance originates from a small fraction of the texture's texels, and if all texels are randomly sampled with equal probability, these small, influential texels will rarely be hit. To reduce this noise, it is crucial to importance-sample these textures.

One such approach to importance-sampling large emissive textures is to use inverse transform sampling. Though it is possible for traditional inverse transform sampling to perform quite well, we have found this sampling performance to be unreliable due to the unpredictable nature of binary search. On the set of high dynamic range images (HDRIs) in Figure 39-1, we found performance of traditional inverse transform sampling to vary dramatically from image to image, by about 10×. Moreover, traditional inverse transform sampling does not allow developers to control whether to prioritize variance reduction over performance or vice versa.

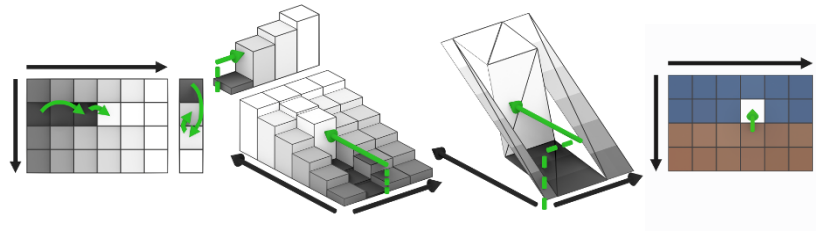


Figure 39-2. *This chapter demonstrates how traditional inverse transform sampling (left) can be reinterpreted as a ray tracing problem through simplified probability geometry (middle) to importance-sample HDRI textures (right). These rays can then be accelerated in hardware, improving performance.*

Our work explores a reformulation of inverse transform sampling that replaces the traditional binary searches performed in texture space with a single hardware-accelerated ray traversal search over cumulative probability geometry. (See Figure 39-2.) By leveraging ray tracing hardware, we are able to improve performance of raw 8K HDRI importance sampling by 1.2–11.2×. Using this new sampling strategy, we show that these importance sampling performance improvements translate to improved rendering performance, by up to 70% on the HDRIs we tested, all while achieving a similar reduction in variance to traditional inverse transform sampling. Moreover, the intersected cumulative probability geometry can be simplified more or less to enable prioritizing performance over variance reduction.

39.2 TRADITIONAL 2D TEXTURE IMPORTANCE SAMPLING

When integrating a function f, the core idea behind importance sampling is to use a modified form of the Monte Carlo estimator, where sampled values $f(x_i)$ are divided by their corresponding probabilities of being sampled:

$$F_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)}. \tag{39.1}$$

This modified estimator enables us to redirect samples toward more influential regions, where $p(x_i)$ effectively serves as a correction factor for that redirection.

To use this modified estimator for texture importance sampling, we first need to rank how influential each texel in our image is, which we do by converting the image from linear RGB to grayscale luminance L. This gives us a single

scalar value per texel that we can use to estimate how much light a particular texel will emit. Next, we consider texels with high luminance to be more important (and vice versa) by computing a joint probability distribution $p(x, y)$ from these luminance values (*joint* here meaning we use both x- and y-coordinates to look up the probability of a texel):

$$p(x, y) = \frac{L(x, y)}{\sum_{i,j} L(i, j)}. \tag{39.2}$$

Once we have this joint probability distribution, the next step is to further break down our probabilities into marginal and conditional probabilities that we can use to importance-sample a row and a column, respectively.

In order to importance-sample a row of interest, we need to compute the *marginal probability* $p(y)$ for each row. This marginal probability describes the overall probability of a row and allows us to sample the rows separately from the columns (typically stored in the *margins* of a joint probability table):

$$p(y) = \sum_{x} p(x, y). \tag{39.3}$$

Then, to importance-sample the column within our previously sampled row, we need to compute the *conditional probability* $p(x|y)$ for each texel within that row. The *conditional* terminology here refers to this probability being conditional on another event happening. In our case, we want to compute the probability of sampling an x given that we already picked a y:

$$p(x|y) = \frac{p(x, y)}{p(y)}. \tag{39.4}$$

At this point, we need to come up with a way to perform our sample redirection. We want to randomly pick from a probability distribution such that the probability of choosing a sample matches that sample's earlier assigned probability. This importance sampling process also needs to be efficient, as we will be taking many of these samples in our ray tracer. This is commonly done through a process called *inverse transform sampling* [5], which "transforms" our probability distributions to enable an efficient sampling algorithm.

More specifically, we will transform the previous probability distribution functions (PDFs) into *cumulative distribution functions* (CDFs). A cumulative distribution function evaluated at some location x returns the probability that a sample will occur whose probability is less than or equal to x. We denote

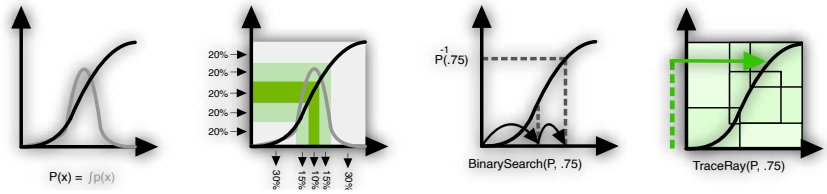


Figure 39-3. *From left to right: by transforming our PDF into a CDF, we can invert this CDF to redirect uniform samples toward highly influential regions. When the CDF cannot be inverted analytically, a binary search can be used to search for the inverted value. Our ray traversal approach leverages the geometric interpretation of inversion to do a similar search in hardware.*

PDFs using a lowercase p and CDFs using an uppercase P:

$$P(x) = \int_0^x p(x)\,dx. \tag{39.5}$$

Because a CDF is just a running sum of positive probability values, these functions only go up in value, and never go down. As a result, a CDF can be inverted. (See Figure 39-3.)

This ability to invert our CDFs is exactly what we need for importance sampling, as this inversion process essentially maps a uniform distribution into a distribution where highly influential samples are more likely to be sampled and low influence samples are less likely to be sampled. In practice, we will need to do this inversion numerically, because our PDFs are actually discrete arrays derived from luminance and do not come from an analytical, invertible equation. Fortunately, this numerical inversion can be done using a fast $O(log(n))$ binary search:

$$P(y) = \int_0^y p(y)\,dy, \qquad P(x|y) = \int_0^x p(x|y)\,dx. \tag{39.6}$$

In practice, we compute and store a single marginal CDF $P(y)$ as well as a conditional CDF $P(x|y)$ for each row, and we discard the original marginal, conditional, and joint PDFs to safe memory. Then, during sampling, we generate our two uniformly random numbers ξ_1 and ξ_2. From here we employ a binary search to find the first y-coordinate from the marginal CDF where $P(y) \geq \xi_1$. Using that y-coordinate, we pick the conditional CDF for that y and again perform a binary search to find the x-coordinate, where $P(x|y) \geq \xi_2$.

We still need to compute our probability of taking this sample, because we need that probability to serve as our correction factor in Equation 39.1. As our CDFs are just a running sum of our original PDFs, we can compute the

original probability densities in x and y by doing a simple subtraction: $p(x|y) = P(x|y) - P(x-1|y)$ and $p(y) = P(y) - P(y-1)$. From there, the joint PDF $p(x, y)$ can be computed as $p(x, y) = p(x|y)p(y)$.

39.3 RELATED WORKS

In this chapter, our goal is to optimize this inverse transform sampling process using ray tracing hardware. Our work takes inspiration from two related works: that by Lawrence et al. [4] and by Cline et al. [3]. Lawrence et al. describe a method that improves on the memory efficiency of large one-dimensional CDFs. To do so, the authors propose using a sparse, piecewise linear approximation of the CDF as a substitute to the previously described dense array of probabilities. The work by Cline et al. goes in a different direction, instead sacrificing memory efficiency for faster sampling performance. Their work identifies the binary search described in Section 39.2 to be a bottleneck and proposes several alternatives to this search that improve performance. (See also Chapter 21.)

39.4 RAY TRACED INVERSE TRANSFORM SAMPLING

The inverse transform sampling method described in Section 39.2 has a near optimal runtime complexity of $O(\log(N) + \log(M))$, where N and M represent our texture dimensions. Still, there are several ways that we can further optimize this strategy to enable more samples to be taken at the same performance budget, while also improving on memory efficiency.

More specifically, the implementation constants associated with inverse transform sampling are still relatively high, especially for real-time and interactive applications. Binary search employs memory access patterns that prevent prefetching of data that is likely to be accessed in the next iteration. The row and column accesses also depend on random numbers, making these accesses incoherent.

To further improve performance and memory efficiency, our technique reduces the size of these implementation constants by *simplifying* the CDFs similar to Lawrence et al. [4], while also using a representation compatible with ray tracing hardware to improve search performance like Cline et al. [3].

Within a CDF, many neighboring probabilities are similar in value. For example, if a texture contains a smooth gradient for the sky, luminance between neighboring pixels will be similar, and as a result, neighboring

probabilities will also be similar. We can rank neighboring pixels' similarity based on how close their corresponding probabilities are. Using that similarity measure, we can prioritize merging highly similar pixels first and avoid merging neighboring pixels with very different probabilities. Structurally, this simplification measure presents us with a trade-off between importance sampling accuracy on one hand and data locality on the other.

In practice, this merging is easier said than done. By merging neighboring values together, our CDFs go from being *structured* to *unstructured*, and we therefore need to devise a strategy to efficiently search through this unstructured data. For this strategy, we further note that the CDFs can be interpreted *geometrically* as height fields. For each address in the CDF array, the associated probabilities can be interpreted as "heights" ranging from 0 to 1. (See Figure 39-2, middle.) From this, we can construct an actual geometric mesh to represent the height field using linear ribbon segments.

With this unstructured, geometric reformulation, we can implement the search required by inverse transform sampling by tracing a ray. To sample the CDF geometry, we again generate a uniform random number $\xi \in [0, 1)$, but rather than searching for that random number using binary search, we instead use that random number to control the height of the ray. That ray is traced toward the CDF height field, where traversal is used to facilitate the search:

$$r = o + \mathbf{d}t, \qquad o = (0, \xi, 0), \qquad \mathbf{d} = (1, 0, 0). \tag{39.7}$$

By randomizing the y offset of the origin and aiming the ray toward the x-axis, the ray is traced toward the positive incline of the height field such that the intersection distance t represents our inverted x-coordinate.

To compute the corresponding sample probability *correction factor* required by our Monte Carlo estimator (Equation 39.1), we need to compute the derivative of the CDF at our sample location, because the derivative of the CDF is our PDF that we want to sample (see Equation 39.5). Fortunately, as our height field geometry consists of linear segments, this derivative can be found by computing the slope of the hit linear segment.

Up until this point, we would need to trace a ray to facilitate the first search over the marginal CDF $P(y)$, followed by a second ray to search over the conditional CDF $P(x|y)$. However, with a few modifications, we can search over both CDFs simultaneously using just one ray.

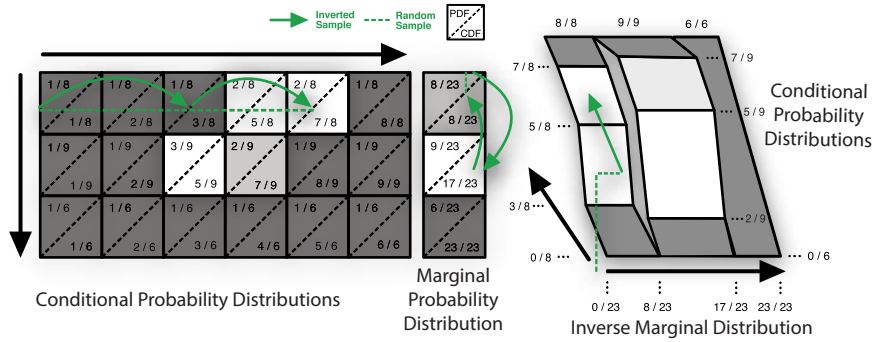


Figure 39-4. *Left: marginal and conditional CDFs and two binary searches are used to importance-sample a row, then a column. Right: we construct geometry that encodes the CDFs on the left (ribbon width and ribbon ID encode the inverse marginal CDF; ribbon height and depth encode the conditional CDFs). Rays can then be traced against this geometry to sample a row and a column, where wider ribbons and taller ribbon segments are more likely to be hit.*

Instead of representing each CDF geometry as separate ribbon geometry, we can leverage the third dimension available to us and the hit group records associated with our ribbons to simultaneously search over the marginal and conditional distributions. Like before, each conditional CDF is represented using simplified ribbon geometry, where the height of the ribbon (*y* in our case) is used to encode the probability for a given row's column (*x* in our case). But now, we can encode the inverse of the marginal *CDF* by altering the thickness and placement of those ribbons along the third dimension (*z* in our case), storing the corresponding "projected" row ID and row probability in the ribbon hit group record that we can look up in a closest-hit program.

By representing the marginal probabilities using ribbon widths and placing these ribbons side by side, we "project" our marginal CDF heights onto the *z*-axis, effectively encoding the inverse of the marginal CDF. To recover the "projected" address for each marginal CDF segment, we look up the hit geometry's corresponding hit record data. Rows that have a *marginal* probability of 0 (meaning that this row's luminance is zero) would obtain zero thickness and thus cannot be hit by rays at all. Conversely, rows that have a high marginal probability have very wide thickness, making those ribbons more likely to be hit. (See Figure 39-4.)

When tracing rays against this *combined* geometry, we keep the direction of the ray oriented toward the CDF's positive incline, $\mathbf{d} = (1, 0, 0)$, but we now set the origin of the ray $o = (0, \xi_1, \xi_2)$, where ξ_2 is the random number that we use

Listing 39-1. *Intermediate CDF geometry control point representation.*

```
struct ControlPoint {
    // Column and row index
    int x, y;
    // The CDF value at the above coordinates
    float f;
    // First-order forward partial derivative in x
    float dfdx;
    // Second-order forward partial derivative in x
    float d2fdx;
};
```

to sample the marginal distribution and ξ_1 is the random number to sample the conditional CDF. From this ray, we obtain the sampled y-coordinate from the marginal distribution through the row ID associated with the ribbon we hit, and then the x-coordinate like before through the returned hit distance t.

39.5 IMPLEMENTATION DETAILS

At this point, we have a set of observations that we can use to implement ray traced inverse transform sampling. Still, some care must be taken when simplifying the conditional and marginal CDFs to preserve variance reduction while also reducing implementation constants. In our implementation, rather than merging neighboring probabilities together at a local scale, we instead collect a subset of texels at a global scale that we consider to be "important." We call these important texels *control points*, as we will use these control points to generate the CDF geometry. Our control point structure looks like the code in Listing 39-1.

As a first step, we construct a control point for each texel in our texture, inserting those control points into a list. For each control point, we store the value of the conditional CDF at that coordinate, as well as the first- and second-order partial derivatives of the conditional CDF with respect to x (the direction of our rows).

Next, we sort this array of control points in descending order by their second-order derivatives. By sorting by the second-order derivative, highly influential, "spiky" control points will appear earlier in the list. Control points with smaller second-order derivative values will appear later in the list, indicating that the neighboring texels of these control points are "similar" and that these control points can be discarded to simplify the geometric representation. We then truncate this sorted list, such that all but N control points are discarded.

These resulting control points represent row-wise CDFs, i.e., the lowest value per CDF is 0 and the highest value is 1. By discarding control points *globally*, however, we might end up with row CDFs that have no control points associated with them at all. To handle this, we first increment the x-coordinate of all control points by 1, making these control points vertex-centered at the ends of the texel grid instead of cell-centered. Next, we enforce that each row is represented by inserting control points at $x = 0$ with a probability of $P(0|y) = 0$ and inserting control points at $x = n$ (if they don't already exist) with a function value of $P(n|y) = 1$.

After sorting and truncating our list of control points by their second-order partial derivatives, we again sort these control points, first by their x-coordinate and then by their y-coordinate, the latter sort being a stable sort to retain the order in x. These operations effectively give us a list of unstructured CDFs sorted by row, with each row being represented with at least two control points.

```
OPTIX_CLOSEST_HIT_PROGRAM(CdfCH)()
{
  auto &lp = optixLaunchParams;
  PRD& prd = owl::getPRD<PRD>();
  const auto &g = owl::getProgramData<CdfGeom>();
  float2 b = optixGetTriangleBarycentrics();
  int id = optixGetPrimitiveIndex();
  // Use left (even id) or right (odd id) triangle edges.
  float alpha = (id & 1) ? 1.f - (b.x + b.y) : (b.x + b.y);
  prd.x = optixGetRayTmax() * lp.environmentMapWidth;
  prd.y = g.rowStart * (1.f - alpha) + g.rowEnd * alpha;
  prd.colPdf = g.triPdfs[id];
  prd.rowPdf = g.geomPdf;
}
```

We now have simplified the CDFs to a point where we can turn the control points into a geometric representation. Our ultimate goal is to trace rays against this geometry, and to improve traversal performance, we use the hardware-accelerated traversal made available with ray tracing cores. So, for each unstructured CDF row, we generate the ribbon geometry described in Section 39.4 such that consecutive CDF control points in a row are connected by triangulated planar quads. And as mentioned earlier, from one row to the next, we adjust the thickness and placement of these ribbons in the third dimension using corresponding per-row marginal CDF values.

We provide a prototypical implementation of our ray inverse sampling technique that is based on OptiX 7 and the Owl Wrapper Library (OWL) [6]. Open source code can be found at the book's source code website and at https://gitlab.com/szellmann/owlcdf.git

Listing 39-2. *An OptiX implementation of ray traced CDF inversion.*

```
__device__ Sample sampleCDF_BVH(float rx, float ry)
{
  auto &lp = optixLaunchParams;
  vec3f org = {0.f, ry, rx};
  vec3f dir = {1.f, 0.f, 0.f};
  PRD prd{-1, -1, -1.f, -1.f};
  Ray ray(org, dir, 0.f, 1.f);
  owl::traceRay(lp.cdf, ray, prd,
                OPTIX_RAY_FLAG_DISABLE_ANYHIT |
                OPTIX_RAY_FLAG_TERMINATE_ON_FIRST_HIT);
  return {prd.x * lp.width, prd.y * lp.height, prd.rowPdf, prd.colPdf};
}
```

As some final, but important optimizations, we normalize the dimensions of the CDF geometry, as we have found this leads to more consistently sized bounding boxes and greatly improved traversal performance, by 45–50% on the data sets we tested. We then correct for this normalization scale after sampling. Finally, if we detect neighboring rows that contain only two control points each, we can further improve performance (by 20–30% for our data sets) by merging these neighboring rows together. As a result, one ribbon can span several rows. We upload the rows spanned by a ribbon through the hit group record of the ribbon geometry, and we use triangle barycentrics to determine which row within the ribbon that ray sampled.

When sampling, we use the code in Listing 39-2.

39.6 EVALUATION

To evaluate ray traced inverse transform sampling, we conducted a series of experiments on an RTX 3090 using a set of ten different high dynamic range environment textures taken from HDRI Haven [7] to illuminate the Amazon Bistro data set [1] (see Figure 39-1). These emissive textures range from 2K to 8K in resolution and contain a mix of *homogeneous* and *heterogeneous* luminance (whether or not a small set of pixels contribute the majority of the overall emitted radiance). These experiments compare how ray traced inverse transform sampling performs against binary search–based inverse transform sampling in terms of general performance improvements, scalability with texture size, and increase in variance due to geometric CDF simplification.

To evaluate performance and scalability, for each texture resolution we recorded two measurements. The first metric measures the theoretical

maximum number of samples that can be inverted per second. This first measure intentionally does not consider external factors involved when rendering a scene, and instead focuses solely on raw sample inversion performance. The second metric measures the total number of samples rendered per second when directly lighting the bistro scene with the environment textures. This second metric uses rays to compute primary visibility information, then performs inverse transform sampling (either ray based or binary search based) to sample from the dome light texture, followed by a shadow ray to compute dome light visibility.

To evaluate the increase in variance that occurs during CDF simplification, we first render a 4000-spp (samples per pixel) image with binary search–based inverse transform sampling. This 4000-spp image serves as a *ground-truth* representation of the scene being lit by the dome light. From there, we measured the mean ꟻLIP error [2] between 32-spp images against our ground truth 4000-spp image. This ꟻLIP metric was chosen as it is specially designed to compute differences between rendered images and corresponding ground-truth images. (See Chapter 19 for more about the ꟻLIP metric.)

All of the these measurements were taken using both a baseline method (binary search–based inverse transform sampling) and our ray traced inverse transform sampling, the latter using a variety of different CDF geometry simplification rates. We then compared relative performance improvements as well as any increase in ꟻLIP error incurred by our approach over the reference method.

39.6.1 MERGING STRATEGY EFFECTIVENESS

To analyze how effective our merging strategy is, we can directly visualize the resulting proxy geometry (see Figure 39-5) for the different textures in Figure 39-1. For heterogeneous textures, we found it most effective to remove 99–99.99% of the original control points, as the proxy geometry only needs to accurately capture a small number of inflection points from the original CDFs. For homogeneous textures, a more aggressive decimation can be used.

In practice, we find our merging strategy to work well for some textures, but it could likely be improved for others. We found the Sun HDRI to be one of the cases where our CDF simplification works best, as the majority of the width of the proxy geometry is allocated to the few influential rows containing the bright sun in the sky. Because there is no other bright light in the same row as the sun, the majority of the vertical height is represented using just a few

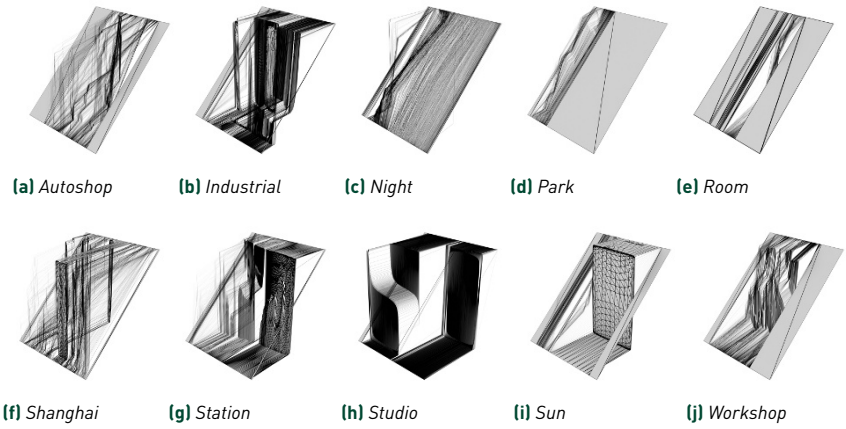


Figure 39-5. *Proxy geometry constructed from different HDRI CDFs, with a decimation rate of 99.9%. Ribbon widths and horizontal placement are used to encode the marginal CDF, and ribbon height at a given depth encodes the conditional CDF for a given row. (For a single-pixel image, this proxy geometry would appear as a fully flat diagonal plane.)*

quads that represent the columns containing the sun. The Park HDRI also works well, as a significant portion of the sky is equally bright, and as a result, many rows are merged together.

However, other HDRIs could likely be simplified much more while still preserving the overall detail (and therefore variance reduction) of the original CDF arrays. At the moment, we currently only merge rows that have been completely simplified. But with textures like the Studio HDRI, there are many influential rows that are likely to be hit that cannot be simplified down to just two control points at the ends because there are two or more influential collections of pixels on the same row. As a result, we are unable to merge these rows together, which will negatively affect ray traversal performance. We would likely benefit from a more general row merging strategy that combines rows with similar height profiles.

39.6.2 PERFORMANCE, SCALABILITY, AND VARIANCE

To determine how this CDF simplification translates to performance and scalability, we first measure our raw performance improvements against the reference method at 2K, 4K, and 8K HDRI resolutions, and at a variety of CDF decimation amounts ranging from 90% to 100% decimation. Results are illustrated in Figure 39-6.

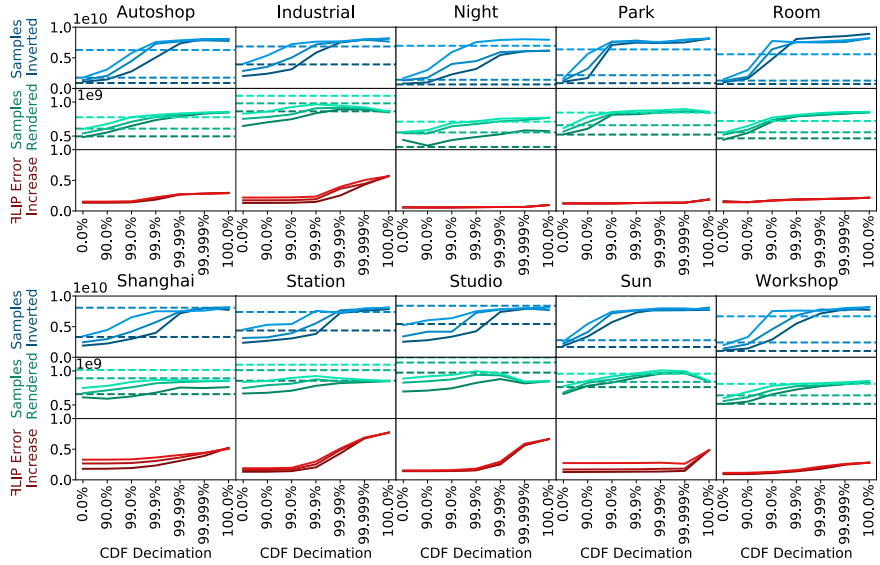


Figure 39-6. *Evaluation of ray traced inversion versus binary search inversion. Dashed lines represent binary search performance, and solid lines represent ray traced inversion performance at various CDF geometry decimation amounts. Dark, medium, and light colors represent 8K, 4K, and 2K HDRIs, respectively. The y-axis magnitude is common across all rows, and the samples inverted, the samples rendered, and the ꟻLIP error increase are* 1×10^{10}, 1×10^{9}, *and* 1, *respectively.*

First, we found that the reference method does surprisingly well for some of our HDRIs. For example, with the Studio texture, the reference method inverts an impressive 5.4 billion samples per second on 8K HDRI. The Station, Industrial, and Shanghai textures also do very well, with an average of 4.3 billion samples inverted per second. However, the other six HDRIs—Workshop, Autoshop, Sun, Night, Room, and Park—achieve on average only 0.97 billion samples inverted per second for the 8K configurations. The reference also noticeably drops in performance when increasing HDRI resolutions from 2K to 8K.

These raw importance sampling results roughly translate to real-world performance, indicating that importance sampling performance is a bottleneck. Five of the 8K HDRIs achieve a high average of 864 million samples rendered per second, and the other five 8K HDRIs achieve a low average of just 468 million samples rendered per second. An interesting exception between synthetic and real-world performance is a rise in performance with the Sun HDRI, perhaps due to more coherency over other HDRIs with respect to the resulting visibility rays.

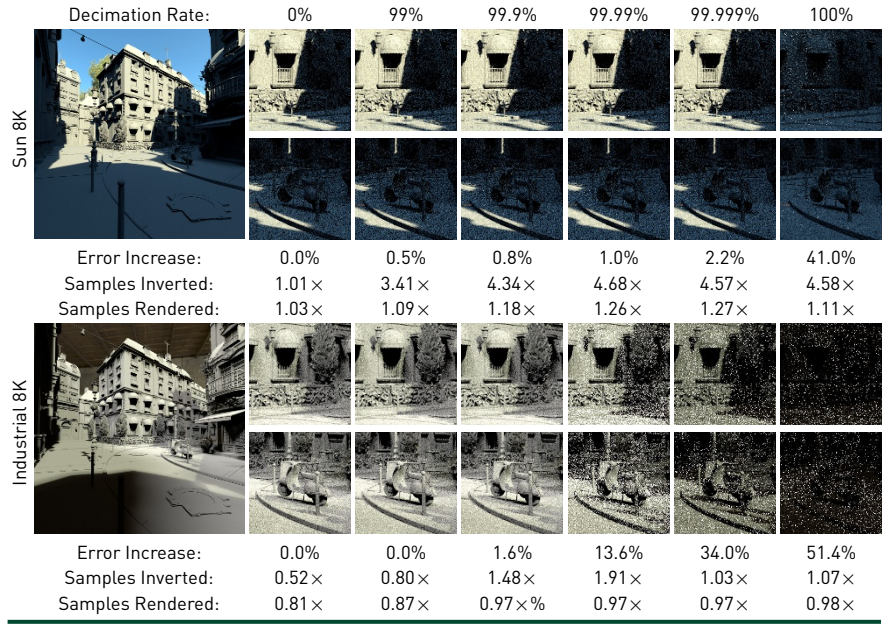


Figure 39-7. *The effect of CDF geometry decimation on noise in a case where we perform well (Sun, top), and a competitive case where binary search performs well and is difficult to beat without introducing variance (Industrial, bottom). A decimation rate of 0% effectively illustrates binary search noise levels and of 100% illustrates naive random sampling noise levels.*

With our ray traced inverse transform sampling, our raw sample inversion benchmark shows that the more we simplify the CDF geometry, the more our performance improves, where we either meet or significantly overcome the number of inverted samples per second compared to the reference. Our method is much more consistent in terms of inversion performance from one HDRI to the next, and it appears to scale much better with respect to texture resolution than the reference does.

As predicted, this performance is heavily dependent on how much the CDF geometry can be decimated without introducing a significant increase in variance. From our evaluations, we are able to decimate the CDF surface up to 99.9% without a noticeable increase in noise for most heterogeneous HDRIs. At 99.99% decimation, we start to see a noticeable increase in noise levels, but still preserve shadows and other structures cast by strong localized light sources while also further reducing importance sampling overhead (see Figure 39-7). For more homogeneous HDRIs, we are able to use a more aggressive decimation without a noticeable impact on variance,

however these HDRIs are less likely to benefit from importance sampling in the first place.

For the six HDRIs with which the reference method struggles, we see significant performance improvements, especially as texture sizes grow larger. For the other four HDRIs for which the reference method performed well, we are slower for 2K textures, approximately equal in performance for 4K textures (albeit more noisy), and faster for 8K textures. Our findings show that when binary search performs well, ray traced inversion also performs well, but runs into performance limits imposed by the overhead of tracing a ray (specifically, the cost of a context switch between the streaming multiprocessor and the ray tracing cores).

We do observe diminishing returns as the CDF is simplified more and more, especially with respect to variance reduction. At a 100% simplification rate, our CDF geometry becomes a single quad, which effectively samples all pixels with equal probability (dramatically increasing variance), but is also much slower than naive random sampling (by about 30%), due to the introduced ray tracing overhead. We have also found ray inverse transform sampling to introduce a small stall for subsequent rays, which adds to this overhead.

For both methods, memory traffic dictates overall performance. For the cases where the reference performs poorly, memory traffic is high, and vice versa. This suggests that some HDRIs have high probability sections near even divisions of the texture, and therefore binary searches are coherent and return quickly, whereas other HDRIs have high probability sections at odd intervals that require many binary search iterations to locate. For homogeneous HDRIs, the reference slows down due to many equally likely pixels causing high memory divergence and many binary search iterations. With ray traced CDF inversion, this memory traffic comes from ray traversal. By simplifying the CDF geometry, we reduce the memory traffic required for CDF inversion.

39.7 CONCLUSION AND FUTURE WORK

In this chapter, we presented a novel technique for inverse transform sampling and its application to large HDRIs. The proposed geometric reformulation of the original problem allows us to greatly simplify the data structure used to store CDFs. This however comes at the cost of turning a structured data representation into an unstructured one. Unstructured data representations are usually more expensive to iterate over. In our case, however, as we iterate through the unstructured data using

hardware-accelerated tree traversal, the implementation constants are relatively low, and in general, the benefits outweigh the potential overhead.

With the proposed implementation, we still see room for improvement. For example, we believe that more careful mesh simplification of the CDF geometry would allow us to reduce the number of triangle primitives even more, further reducing bounding volume hierarchy traversal overhead. We would also find it interesting to test if our approach is effective not just for 2D CDF inversion, but also for more general 1D CDFs. Finally, we see a number of different use cases to which our method could be applied or extended, for example, emissive textures mapped on triangle meshes or emissive voxels in a volume.

REFERENCES

[1] Amazon Lumberyard. Amazon Lumberyard Bistro. *Open Research Content Archive (ORCA)*, http://developer.nvidia.com/orca/amazon-lumberyard-bistro, 2017.

[2] Andersson, P., Nilsson, J., Akenine-Möller, T., Oskarsson, M., Åström, K., and Fairchild, M. D. ꟻLIP: A difference evaluator for alternating images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2):15:1–15:23, 2020. DOI: 10.1145/3406183.

[3] Cline, D., Razdan, A., and Wonka, P. A comparison of tabular PDF inversion methods. *Computer Graphics Forum*, 28(1):154–160, 2009. DOI: 10.1111/j.1467-8659.2008.01197.x.

[4] Lawrence, J., Rusinkiewicz, S., and Ramamoorthi, R. Adaptive numerical cumulative distribution functions for efficient importance sampling. In *Proceedings of the Sixteenth Eurographics Conference on Rendering Techniques*, pages 11–20, 2005.

[5] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory To Implementation*. Morgan Kaufmann, 3rd edition, 2016.

[6] Wald, I., Morrical, N., and Haines, E. OWL: A node graph "wrapper" library for OptiX 7. https://github.com/owl-project/owl, 2020.

[7] Zaal, G. HDRI Haven. https://hdrihaven.com/, 2016.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 40

ACCELERATING BOOLEAN VISIBILITY OPERATIONS USING RTX VISIBILITY MASKS

Dirk Gerrit van Antwerpen and Oliver Klehm
NVIDIA

ABSTRACT

When inspecting or presenting detailed scenes, it is often necessary to clip away parts of the geometry to get a better view of the internals of a model. Although clipping can be performed directly on the geometry, this is a costly and complex operation, often preventing interactive visualizations. In this chapter we'll discuss how to accelerate complex Boolean visibility operations using the NVIDIA OptiX API. The method presented here does not require any preprocessing of the geometry and supports arbitrary Boolean clipping operations using arbitrary visibility shapes in real time. All ideas presented in this chapter apply equally well to the DirectX Raytracing and Vulkan Ray Tracing APIs.

40.1 BACKGROUND

NVIDIA OptiX allows applications to attach custom `anyhit` programs to geometry. The `anyhit` program is invoked whenever a ray intersects the

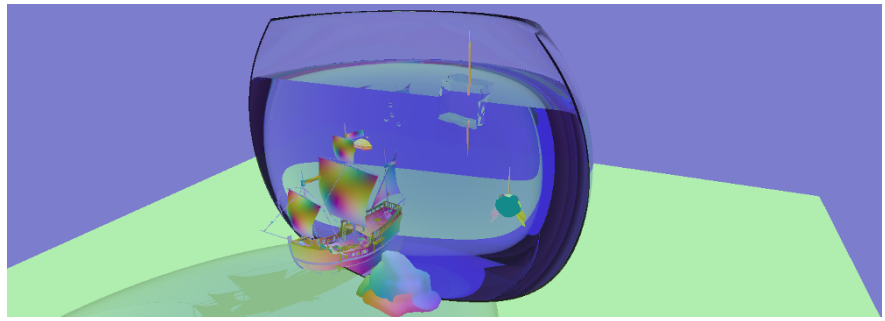


Figure 40-1. *Fishbowl scene with varying Boolean clipping operations applied to its geometry.*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_40

geometry. The `anyhit` programs may choose to skip the intersection by calling `optixIgnoreIntersection`. These programs can thus be used to implement cutouts and partial visibility of geometry. Although flexible, custom `anyhit` program invocations interrupt the fixed-function RTX ray traversal, which comes with considerable overhead [1].

Orthogonally OptiX also provides access to a fixed-function RTX-hardware-accelerated 8-bit visibility mask (see documentation [3, 2]). An application is able to assign a custom visibility mask to each OptiX instance. Similarly, the application assigns a custom ray visibility mask to each traced ray by passing the mask as an argument to the `optixTrace` call. If for any instance the visibility condition `(ray.mask & instance.mask) != 0` is false, the instance is culled during traversal.

The visibility mask can thus be used to specify which instances will be intersected along a ray. The 8 bits in the masks can be used to encode eight independent visibility groups. Each instance is assigned to any or all of the groups using the instance visibility mask at acceleration build time. The per-ray visibility mask passed to `optixTrace` specifies which groups are to be visible to this ray. For a given `optixTrace` call, each instance is either fully visible or fully invisible. Thus, it is not possible to express partial visibility, as is required for clipping, using the masks directly.

40.2 OVERVIEW

In this chapter we will build on the hardware-accelerated visibility masks to implement an efficient method for partial visibility of instances. We will specify partial visibility of instances using arbitrary visibility shapes and Boolean-valued visibility functions. Using these, we will implement a higher-level trace operation to intersect a ray against partially visible instances by combining multiple `optixTrace` calls and evaluating the Boolean-valued visibility function inside an `anyhit` program. We will then show how to accelerate the trace operation by eliminating `anyhit` program invocations using hardware-accelerated instance and ray visibility masks. We give a practical example showing how to accelerate arbitrary Boolean-valued visibility functions by approximating them as disjunctions over eight arbitrary, hardware-accelerated Boolean-valued visibility expressions. Finally, we discuss how to resolve materials on the caps of nested, partially clipped solid instances.

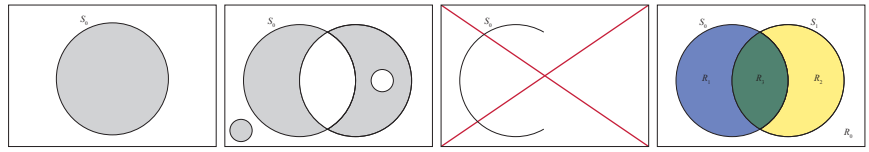


Figure 40-2. *A shape partitions space into well-defined inside and outside regions. Multiple shapes together partition the space into multiple regions. Left: simple shape. Center left: complex shape, but inside and outside regions are well defined. Center right: invalid shape due to ill-defined inside and outside regions. Right: two shapes partitioning space into four regions.*

40.3 PARTIAL VISIBILITY

In this section we will show how to specify partial visibility of instances throughout space. In its most general form we can define visibility for an instance as any arbitrary Boolean-valued function that depends on the position in space. In regions of space where the instance is visible, the function evaluates to `true`, otherwise to `false`. In our method these regions of constant visibility are specified directly by partitioning the 3D space using up to N shapes. Each shape surface is defined by an instance (usually different from the instance whose visibility is to be defined). The surface must partition all space into well-defined inside and outside regions. That is, each shape must be an orientable manifold without boundaries (see Figure 40-2). Together the N shapes partition the 3D space into up to 2^N regions. Each region is exactly defined as the set of shapes of which the region is inside. In other words, any subset of shapes implicitly defines a region. Because visibility of an instance is constant within a region, the instance is visible either everywhere in the region or nowhere. Given these regions of visibility, we can express the visibility function as a Boolean-valued function over the set of shapes of which a region is inside.

40.4 TRAVERSAL

In the last section we specified instance visibility using a set of shapes and a Boolean-valued visibility function taking a subset of shapes as the argument. We now show how to build a higher-level trace operation that respects instance visibility. At first we use an `anyhit` program to evaluate the final visibility of each instance. We will further accelerate this operation using visibility masks and avoid the invocation of `anyhit` programs in the next section.

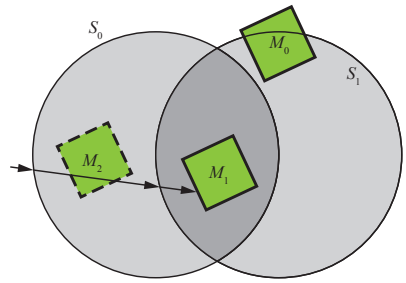


Figure 40-3. *A ray march through three regions before the nearest visible hit is found. The scene has two shapes S_0 and S_1 and three objects M_0, M_1, and M_2 with corresponding visibility functions $V_0 = \neg S_0$, $V_1 = S_1$, and $V_2 = S_1$.*

Our trace operation ray-marches through regions, from one shape surface to the next until it hits visible geometry (see Figure 40-3). Hence, we repeatedly ray-trace against the shape surfaces as well as scene instances. While ray-marching through regions, our algorithm tracks the set of shapes S of which the current region is inside. When the ray hits a shape surface as closest intersection, we update the set S, adding or removing the corresponding shape, before continuing traversal through the adjacent region. For efficiency, we track S using a bit vector of size N, one bit per shape. When the ith bit in the vector is set, shape S_i is in S. This practically limits the number of shapes to some fixed N, but it makes adding or removing a shape to S as easy as flipping a bit in the vector (see Listing 40-1).

Listing 40-1. *Ray marching loop through regions with constant visibility.*

```
// March through regions.
while( true )
{
  // Find the nearest shape surface (region boundary) or visible hit point.
  {instance,t} = optixTrace( ... tmin, tmax, /*visibilityMask*/ 0xFF, ...,
                             /*payload*/ shapeSet );

  if( miss )
    return false; // Report miss.

  if( instance.isSolid )
    return true; // Report hit.

  // We hit a shape surface; update the shape set.
  shapeSet ^= (1u << instance.shapeIndex);
  // Continue in adjacent region.
  tmin = t;
}
```

Listing 40-2. *Visibility* `anyhit` *program.*

```
__global__ void __anyhit__trace()
{
  // Load current shape set from payload.
  const unsigned int shapeSet = optixGetPayload_0();

  // Perform arbitrary software visibility test for current instance using
       shape set.
  if( !instance.evalVisibilityFunction( shapeSet ) )
    optixIgnoreIntersection(); // Skip invisible hit and return.

  // ...
}
```

Finally, the visibility of an instance is determined inside an `anyhit` program by evaluating the instance's visibility function for the current shape set S (see Listing 40-2). The shape set bit vector is passed to the `anyhit` program as part of the data payload of the ray. As the shape set S is constant within each region by construction, so too will the visibility of each instance be constant within a region. Note that the visibility function of an instance can be altered using the Shader Binding Table (SBT) either by indexing to a different `anyhit` program for its geometry or by parameterization of the visibility function using instance-specific data in the SBT record. Refer to Chapter 15 and the OptiX documentation [3] for an in-depth discussion on how to set up the SBT.

40.5 VISIBILITY MASKS AS BOOLEAN VISIBILITY FUNCTIONS

So far, we have ignored the visibility masks altogether and instead opted to evaluate instance visibility in `anyhit` programs. Although flexible, this can quickly become very costly, as our method may invoke many such `anyhit` calls when large parts of the scene geometry are invisible. To accelerate our method, we want to cull invisible instances before their `anyhit` program is even called by leveraging instance and ray visibility masks.

We can interpret the OptiX visibility mask test as the evaluation of a Boolean visibility function V'_i over eight Boolean variables $b_0, \ldots, b_7$. An instance is culled when its visibility function evaluates to `false` under the current variable truth assignment. The ray's visibility mask passed to `optixTrace` encodes the truth assignment of these eight variables, one bit per variable. The instance's visibility mask encodes the instance's Boolean visibility function as a logical disjunction (logical *OR* of Boolean variables); variables appear in the disjunction when their bit is set. For example, the instance visibility mask $0b00010001$ encodes the visibility function $b_0 \vee b_4$.

Listing 40-3. *Use visibility mask to accelerate visibility tests.*

```
// Re-evaluate the Boolean variables b0...b7.
visibilityMask = B(shapeSet);

// Find region boundary or nearest visible hit point.
{instance,t} = optixTrace( ... tmin, tmax, visibilityMask, ...,
                           /*payload*/ shapeSet );
```

We can use this to accelerate the instance visibility function V_i for some instance i by specifying a conservative accelerated visibility function V'_i using the visibility masks. The accelerated visibility function V'_i is conservative with respect to V_i if it never culls the instance when it should be visible; it may however fail to cull invisible instances (false positive). That is, $V_i(S) \Rightarrow V'_i(B(S))$, where B is some arbitrary mapping of the shape set S to the eight Boolean variables $b_0, \ldots, b_7$ making up the ray visibility mask (see Listing 40-3). Note that although the visibility function V_i and the accelerated visibility function V'_i are specific to each instance, the mapping function B is not and instead is globally defined. The choice of B determines how well the accelerated visibility functions can approximate the instance visibility function. Any false positives will incur the overhead of the `anyhit` program.

The simplest such mapping B is the identity mapping B_I, where we just use the current shape set directly as the visibility mask. That is, the Boolean variable b_i is `true` when the current region is inside shape S_i. This allows for partial visibility, however it limits the number of shapes to eight. Furthermore, it limits visible regions to be unions of shapes. We can only approximate intersections of shapes and cannot at all express negations of shapes. See Table 40-1 for examples of instance visibility functions that cannot be (exactly) accelerated under the identity mapping.

Visibility Function	Accelerated under B_I	Accelerated under B_E
$V_0 = \neg S_0$	None	$V'_0 = E_0$
$V_1 = S_0 \wedge S_1$	$V'_1 = S_0$ or $V'_1 = S_1$	$V'_1 = E_1$
$V_2 = S_0 \wedge (S_1 \vee S_2)$	$V'_2 = S_0$ or $V'_2 = S_1 \vee S_2$	$V'_2 = E_1 \vee E_2$
$V_3 = \neg S_0 \vee S_1 \vee S_2$	None	$V'_3 = E_0 \vee E_1 \vee E_2$

Table 40-1. *Example visibility and corresponding accelerated visibility functions under B_I and B_E using expressions $E_0 = \neg S_0$, $E_1 = S_0 \wedge S_1$, and $E_2 = S_0 \wedge S_2$. Under mapping B_I functions V_0 and V_3 cannot be accelerated, whereas functions V_1 and V_2 can only be approximated. Under mapping B_E all functions can be fully accelerated by carefully choosing the accelerated expressions E_i. See Figure 40-4 for an example scene using these visibility functions.*

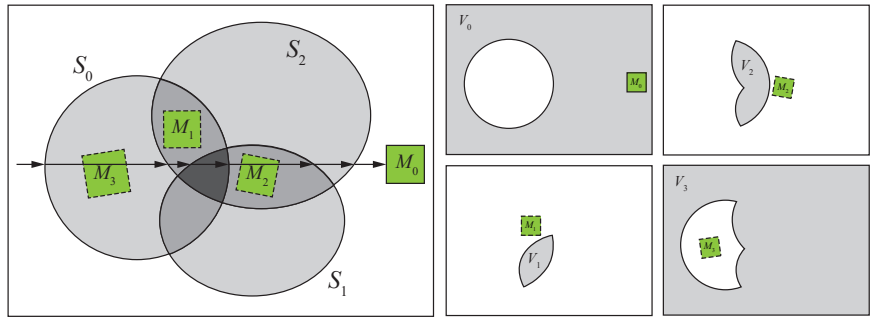


Figure 40-4. *Left: ray traversal through a scene with three shapes and four objects with visibility functions using conjunctions and negations of shapes: $V_0 = \neg S_0$, $V_1 = S_0 \wedge S_1$, $V_2 = S_0 \wedge (S_1 \vee S_2)$, and $V_3 = \neg S_0 \vee S_1 \vee S_2$. Right: visualizations of the four objects and their visibility functions, with the gray areas depicting where the visibility functions evaluate to* `true`.

Instead, we propose a more flexible mapping B_E, where each variable b_i is defined by an arbitrary logical expression E_i over the shapes. Each instance visibility function V_i is a disjunction over a subset of these expressions. This disjunction can be perfectly represented by the accelerated visibility function V'_i. Because the mapping B is only evaluated in software at shape surfaces, these logical expressions are not limited in functionality by the hardware. They can therefore also include conjunctions and negations of the shapes, operations that are not natively supported by the OptiX API (see Figure 40-4). In other words, visibility functions are fully hardware accelerated as long as we can express them all as a disjunction of eight shared accelerated Boolean expressions, where these eight expressions can be freely chosen. Note that if some visibility functions can only be conservatively approximated as a disjunction of these expressions, then those visibility functions are still partially hardware accelerated but will rely on the `anyhit` program to perform the exact visibility test. See Table 40-1 for an example of how expressions can be used to express the visibility functions from Figure 40-4. Table 40-2 shows the corresponding step-by-step evolution of the internal state of the algorithm for the traversed ray from Figure 40-4 using the B_E mapping.

40.6 ACCELERATED EXPRESSIONS

In the last section we introduced the accelerated Boolean expressions. Here, we provide one efficient way of defining and evaluating these expressions for a low shape count N. We take advantage of the bit vector encoding of the shape

S	shapeSet	visibilityMask	Expressions	Objects
$\{\}$	000	001	E_0	M_0
$\{S_0\}$	001	000	None	None
$\{S_0, S_2\}$	101	100	E_2	M_2, M_3
$\{S_0, S_1, S_2\}$	111	110	E_1, E_2	M_1, M_2, M_3
$\{S_1, S_2\}$	110	001	E_0	M_0
$\{S_2\}$	100	001	E_0	M_0
$\{\}$	000	001	E_0	M_0

Table 40-2. *The evolution of the algorithm as the ray in Figure 40-4 marches through regions until a hit is found. The scene uses the visibility functions and B_E mapping from Table 40-1. For each traversed region we show the shape set S, the variables `shapeSet` and `visibilityMask` from Listing 40-3, the accelerated expressions evaluating to `true`, and the resulting set of visible objects.*

set S. The truth table for any arbitrary logical expression over N shapes has size 2^N. We use a 2^N-bit vector to express the truth table of an instance visibility function V_i. The bit vector encoding for S can be used easily to index into the truth table and look up the visibility assignment for the current shape set. This method is only practical for low N, but is very efficient to evaluate. From here on, we use $N = 5$ so we can express a truth table using a single 32-bit word. Listing 40-4 shows the implementation of this visibility test in the `anyhit` program. For larger N the set S and the expression truth tables would be better tracked and defined in sparse formats, at the cost of more complex evaluation. We still need to factor the instance visibility functions into a disjunction of the given accelerated expressions E_j. We again use a 2^N-bit

Listing 40-4. *Fallback visibility `anyhit` program.*

```
__global__ void __anyhit__trace()
{
  // Load current shape set from payload.
  const unsigned int shapeSet = optixGetPayload_0();

  // Expand shape set to standard unit bit vector (i.e., exactly 1 bit set).
  unsigned int shapeSetMask = 1 << (31 - shapeSet);

  // Perform software visibility test.
  if( ( instance.assignment & shapeSetMask ) == 0 )
    optixIgnoreIntersection(); // Skip invisible hit and return.

  // ...
}
```

vector to express the truth table of each expression E_j. The visibility function truth table is combined with the expression truth tables to factor the visibility function (see Listing 40-5). Note that we added a tautology as expression E_7. This is needed to guarantee that any visibility function can at least be conservatively approximated by that expression.

What's left is the implementation of the mapping B_E itself. We use a similar approach of using the shape set to index into truth tables to map the shape set to an OptiX visibility mask (see Listing 40-6).

We didn't specify how to choose the optimal set of expressions to represent all instance visibility functions present in the scene. This highly depends on the application and the type of visibility functions used. In general, finding the optimal set of expressions has NP time complexity. If the visibility functions are unknown at compile time and the application needs to select the expressions at runtime, a greedy approximation could be used to quickly find a reasonable set of accelerated expressions. One such simple approximation is to naively select the 7 most common visibility functions and one fallback tautology as the 8 accelerated expressions. Another straightforward approach selects the 3 shapes appearing most frequently as literals in the visibility functions and use the $2^3 = 8$ unique logical combinations of these literals as accelerated expressions. All other shapes will trigger the `anyhit` software fallback. More advanced approaches would try to identify and exploit common logical relations between literals in the visibility functions.

40.7 SOLID CAPS

Using the shapes and visibility functions, we can efficiently skip intersections with clipped instances. However, it may also be required to shade solid instances that are only partially clipped away. Up to this point, the surfaces of such instances would simply be partially visible and solid geometry would appear to be hollow. That is, no cap will appear where a shape surface cuts through partially visible solid instance geometry. Optionally, it is possible to record hits on such a cap (see Figure 40-5). A cap is always part of a shape surface. A hit point on a shape surface is part of a cap if the hit point lies inside a solid instance that is visible on one side of the shape surface, but invisible on the other side. In other words, the visibility of the solid instance flips at the shape surface. We can check for any instance if its visibility changes by evaluating the visibility function using the shape sets on either side of the surface (see Listing 40-7).

Listing 40-5. *Setup of instance visibility masks for the objects in Figure 40-4.*

```
// Generic 32-bit masks for 5 shapes (only 3 are used in this sample)
const unsigned int MASK_TRUE = 0b11111111111111111111111111111111u;
const unsigned int MASK_S0   = 0b01010101010101010101010101010101u;
const unsigned int MASK_S1   = 0b00110011001100110011001100110011u;
const unsigned int MASK_S2   = 0b00001111000011110000111100001111u;
const unsigned int MASK_S3   = 0b00000000111111110000000011111111u;
const unsigned int MASK_S4   = 0b00000000000000001111111111111111u;

// Array of truth tables of all accelerated expressions
const unsigned int masks[8] = {
 ~MASK_S0,           // E0 = ¬S0
  MASK_S0 & MASK_S1, // E1 = S0 ∧ S1
  MASK_S0 & MASK_S2, // E2 = S0 ∧ S2
  // ...
  // Add tautology to guarantee that all functions can be approximated.
  MASK_TRUE          // E7 = 1
};

// Map a truth table mask to a disjunction of RTX-accelerated expressions.
void setupInstance( Instance &instance, unsigned int assignment )
{
  unsigned int visibilityMask = 0;
  unsigned int joinedMask = 0;

  // Enable all implying clauses.
  for( unsigned int i = 0; i < 8; ++i )
  {
    // masks[i] -> assignment
    if( ( ~assignment & masks[i] ) == 0 )
    {
      joinedMask     |= masks[i];
      visibilityMask |= ( 1 << i );
    }
  }

  // Check for biconditional.
  if( joinedMask != assignment )
  {
      // Instance visibility functions cannot be exactly expressed using
           accelerated expressions. Set tautology expression and force
           anyhit program for software fallback.
    visibilityMask |= ( 1 << 7 );
    instance.flags |= OPTIX_INSTANCE_FLAG_ENFORCE_ANYHIT;
  }

  instance.assignment     = assignment;
  instance.visibilityMask = visibilityMask;
}

// ...
// Try to express instance visibility functions as disjunction of RTX-
      accelerated expressions.
setupInstance( M0, ~MASK_S0 );                      // V0 = ¬S0
setupInstance( M1, MASK_S0 & MASK_S1 );             // V1 = S0 ∧ S1
setupInstance( M2, MASK_S0 & (MASK_S1 | MASK_S2) ); // V2 = S0 ∧ (S1 ∨ S2)
setupInstance( M3, ~MASK_S0 | MASK_S1 | MASK_S2 );  // V3 = ¬S0 ∨ S1 ∨ S2
```

Listing 40-6. *Mapping of shape set to visibility mask using arbitrary logic expressions.*

```
// Map shape set encoding to RTX visibility mask
// using accelerated expressions.
unsigned int B_E( unsigned int shapeSet )
{
  unsigned int visibilityMask = 0;

  // Expand shape set to standard unit bit vector (i.e., exactly 1 bit set).
  unsigned int shapeSetMask = 1 << (31 - shapeSet);

  // Enable all conjunctions with matching visibility.
  for( unsigned int i = 0; i < 8; ++i )
  {
    // If shape set implies expression, enable conjunction.
    if( ( shapeSetMask & masks[i] ) != 0 )
    {
      visibilityMask |= ( 1 << i );
    }
  }

  return visibilityMask;
}
```

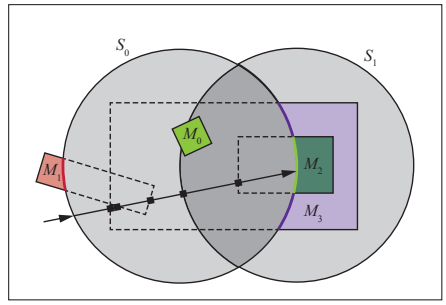


Figure 40-5. *Record the hit on the cap of a nested, partially visible object and resolve the cap material. Ray segments in each region are retraced to update the material stack. The cap hit on shape S_0 resolves to material M_2. The object visibility functions are $V_0 = S_0 \vee S_1$, $V_1 = \neg S_0$, $V_2 = (\neg S_0 \wedge S_1) \vee (S_0 \wedge \neg S_1)$, and $V_3 = \neg S_0 \wedge S_1$.*

We can now check whether a surface hit point is a cap for an enclosing solid instance, but we still need a list of all candidates enclosing invisible solids at the current hit point. We track this using a material stack [5]. Whenever a ray enters a solid instance, the instance is pushed onto the material stack. Whenever a ray leaves a solid instance, the instance is removed from the stack. Note that besides visible solids we also need to track any invisible solids on this stack. Unfortunately all invisible solids have been culled during traversal, so we don't know if we entered or left any. To resolve this, we retrace every ray segment through a region during the ray march (see

Listing 40-7. *Test if the intersection between a solid and a shape instance requires a cap.*

```
bool isCap( Instance &solidInstance, Instance &shapeInstance, unsigned int
    shapeSet )
{
  unsigned int preShapeSet  = shapeSet;
  unsigned int postShapeSet = shapeSet ^ shapeInstance.shapeMask;

  // Expand shape set to standard unit bit vector (i.e., exactly 1 bit set).
  unsigned int preShapeSetMask  = 1 << (31 - preShapeSet);
  unsigned int postShapeSetMask = 1 << (31 - postShapeSet);

  bool preVisible, postVisible;
  preVisible  = (solidInstance.assignment & preShapeSetMask)  != 0;
  postVisible = (solidInstance.assignment & postShapeSetMask) != 0;

  return preVisible != postVisible;
}
```

Listing 40-8). During the retrace, all invisible geometry is forced visible. We use another `anyhit` program to update the material stack at every hit point along the retraced ray segment. Note that the ray segment by construction only intersects invisible geometry. The OptiX API makes no guarantee about the order in which `anyhit` programs will be called for intersections along a ray. Therefore, simply pushing/popping the materials of invisible objects on/off the material stack in the order the programs are called may lead to incorrect nesting of solids. To resolve this, we also track the farthest recorded hit distance along the ray for all solids on the material stack. This hit distance is used to resolve the nesting of solids, so we can shade the cap with the material of the last entered solid (see mesh M_2 in Figure 40-5). Alternatively, one could use user-assigned priorities to resolve the nesting of solids [4].

The resolution of the correct material on caps comes at a significant cost. Every ray segment is retraced, which doubles the number of trace calls. The `anyhit` calls for invisible solids further increase the cost of the retrace calls. However, using `anyhit` programs is far more lightweight than the alternative of ray marching through all invisible hits, which would add another trace call for each crossed invisible surface. Note that it's not feasible to easily merge the trace and retrace calls into a single trace call. The material stack should only be updated for invisible hits up to the next shape surface hit. However, as said, the OptiX API makes no guarantee about the order in which `anyhit` programs will be called, and so it may call the `anyhit` program for invisible hits beyond the next shape surface hit, resulting in an invalid material stack at the shape surface hit. Retracing the exact ray segment up to the shape surface hit resolves this issue.

Listing 40-8. *Ray marching with cap detection.*

```
// March through regions.
while( true )
{
  // Re-evaluate the Boolean variables b0...b7.
  visibilityMask = B(shapeSet);

  // Find region boundary or nearest visible hit point.
  {instance,t} = optixTrace( ... tmin, tmax, visibilityMask | /*fallback*/ 0
       x80, ..., /*payload*/ shapeSet );

  if( !miss )
  {
    // Retrace ray segment to update material stack in anyhit program.
    {instance,t} = optixTrace( ... tmin, t, /*forced visibility*/ 0xFF,
         OPTIX_RAY_FLAG_ENFORCE_ANYHIT, ... );
  }

  if( miss )
    return none; // Report miss.

  if( instance.isSolid )
    return instance.solid; // Report solid hit.

  // We hit a shape surface.
  lastT    = 0;
  capSolid = none;

  // Search last entered cap solid on the material stack.
  for( all solids on material stack )
  {
    // If the solid on the stack is a cap ...
    if( isCap( solid.instance, instance, shapeSet )
    {
      // ... and is farthest along the ray,
      if( solid.lastT > lastT )
      {
        // ... record the cap hit.
        lastT = solid.lastT;
        capSolid = solid;
      }
    }
  }

  if( capSolid is not none )
    return capSolid; // Report solid cap hit.

  // Update shape set.
  shapeSet ^= (1u << instance.shapeIndex);
  // Continue in adjacent region.
  tmin = t;
}
```

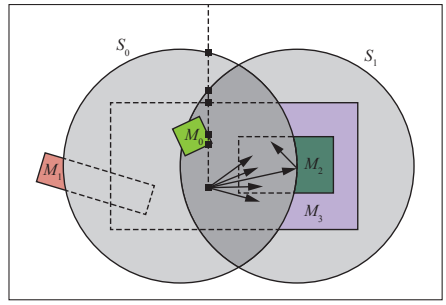


Figure 40-6. *The material stack and shape set for primary rays originating from a camera inside the scene are initialized by tracing a virtual ray from outside the scene to the origin of the ray.*

40.8 CAMERA INITIALIZATION

So far, we have assumed that all ray origins are placed outside of the scene. For such rays the shape set *S* and material stack can be initialized as empty. In practice, however, rays may initiate from anywhere inside the scene, possible even from within invisible geometry. Secondary outgoing rays spawned from a shaded hit point can simply continue with the final material stack and shape set *S* from the incoming ray (see Figure 40-6). However, primary rays need special initialization. A robust approach to generate a valid material stack and shape set is to traverse a setup ray from outside the scene toward the camera origin. During traversal, the shape set and the material stack are updated as usual, but all geometry is considered invisible. The final result is the correct shape set and material stack for the camera origin (see Figure 40-6). Note that the order in which shape surface hits are reported has no effect on the final shape set *S* and we already know the exact endpoint of the ray. Therefore, we don't have to ray-march through the regions for the setup ray, and instead we can update both the shape set and the material stack using `anyhit` programs in a single `optixTrace` call.

Note that primary camera rays often start at (for pinhole cameras) or near (for finite aperture cameras) a common camera origin. We can exploit this by generating a single initial material stack and shape set for this camera origin in a separate pass. We can share these for all primary rays starting from the camera. For finite aperture cameras we still need to trace an extra virtual ray from the common camera origin to the actual starting point of the primary ray on the lens. However, this will generally be a short ray and thus a far cheaper operation than generating the initial material stack and shape set afresh for

each primary ray. Similarly, for an animated camera the initial material stack and shape set for the camera origin can be updated cheaply between frames by tracing a single virtual ray from the old to the new camera origin instead of regenerating these from scratch for each frame.

REFERENCES

[1] Dunn, A. Tips and tricks: Ray tracing best practices. *NVIDIA Developer Blog*, https://developer.nvidia.com/blog/rtx-best-practices/, May 20, 2019. Accessed March 2, 2021.

[2] Microsoft. Instance masking. *DirectX Raytracing (DXR) Functional Spec*, https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html#instance-masking, 2018. Accessed March 2, 2021.

[3] NVIDIA. NVIDIA OptiX 7—Programming guide. https://raytracing-docs.nvidia.com/optix7/guide/index.html, 2021. Accessed March 2, 2021.

[4] Schmidt, C. and Budge, B. Simple nested dielectrics in ray traced images. *Journal of Graphics Tools*, 7(2):1–8, 2002. DOI: 10.1080/10867651.2002.10487555.

[5] Wächter, C. and Raab, M. Automatic handling of materials in nested volumes. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 139–148. Apress, 2019.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 41

PRACTICAL SPATIAL HASH MAP UPDATES

Pascal Gautron
NVIDIA

ABSTRACT

Spatial hashing is a very efficient approach for storing sparse spatial data in a way that easily allows merging information among areas with similar properties. Though having a very simple core, this technique requires careful implementation to achieve high efficiency. In this chapter we first consider some key principles and implementation aspects. The original spatial hashing is initially limited to storing simple data types, which can be updated using atomic calls. The second part of this chapter adds support for arbitrary data storage in the hash map, while still guaranteeing the atomicity of the updates. We achieve this using on-the-fly generation of change lists for each modified hash map entry.

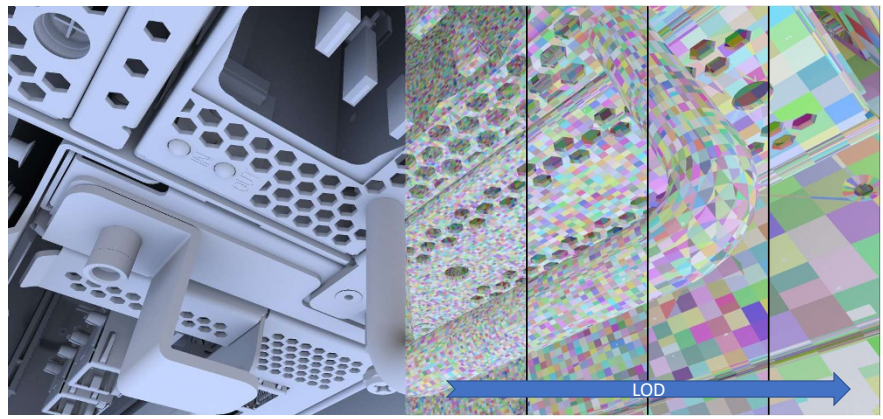


Figure 41-1. *Spatial hashing provides a simple means of storing and retrieving spatial information in constant time, at arbitrary resolutions.*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_41

41.1 INTRODUCTION

The spatial hashing approach [2] is very efficient to store and access sparse spatial data in a massively parallel setting. This is particularly useful for computing and storing lighting information independently from the scene representation. Using variations on the hash function, a spatial hash map can be extended to support multiresolution and dynamic scenes [4].

Despite its efficiency, spatial hashing suffers from a major limitation on the data stored in the hash map. As many threads may simultaneously insert and modify information in the map, each hash map entry has to be updated atomically. Though this can be trivially achieved for simple data structures, atomicity cannot be guaranteed on more complex data without introducing prohibitive costs.

After introducing key aspects for implementing the spatial hashing scheme, this chapter covers an efficient method for parallel updates of arbitrary data structures within the hash map. Change requests are stored into a set of per-entry linked lists, which are then processed in parallel.

41.2 SPATIAL HASHING

Many light transport algorithms store and interpolate lighting information using a world-space representation such as kD-trees [6], octrees [10], or surface subdivision [5]. However, such data structures usually involve nontrivial representations and update algorithms, making them challenging to use in massively multithreaded contexts. Screen-space methods [1, 3] store information per-pixel for improved efficiency. However, they can only represent a subset of the full 3D scene.

Spatial hashing [2] builds upon the idea of hash maps using a hash function designed so that hashing neighboring world-space points results in the same hash index. In other words, spatial hash entries represent sub-volumes of the scene (Figure 41-1). The hash map can be seen as a sparse voxel representation with trivial data structure and constant-time access.

41.2.1 ENTRY ALLOCATION

The spatial hash function H is based on successively applying a simple, 1D integer hash function h on each coordinate of a 3D point P:

$$H(P) = h\left(\lfloor P_z/d \rfloor + h\left(\lfloor P_y/d \rfloor + h(\lfloor P_x/d \rfloor)\right)\right), \tag{41.1}$$

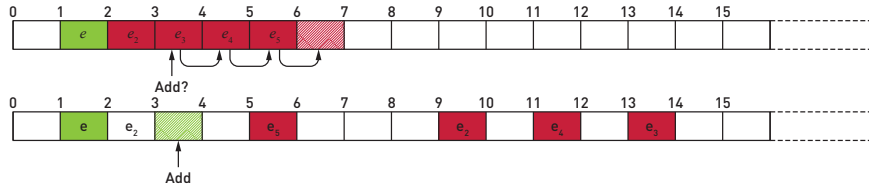


Figure 41-2. *Using linear search, colliding entries mapping to the entry e are stored as e_n in the neighboring entries, hence creating a dense block. Adding another entry in the same area results in many search steps before finding a free slot (top). Using rehashing, the colliding entries are scattered throughout the map, thus reducing the collision rate and preserving the hash map uniformity (bottom).*

where d is a discretization step (i.e., the voxel size). The choice of the integer hash function h is arbitrary, such as the Wang hash [9]. An algorithm can then store light transport data for the related volume at the index $H(P)$ mod s, where s is the number of entries in the hash map.

Collisions are detected by computing and storing a second hash index, or *checksum*, using Equation 41.1 with another integer hash function such as a XOR shift [7]. When adding a new entry into the hash map, we first compute its hash index using the primary hash function. We then generate its checksum and compare it to the checksum stored at the hash index. If the entry is not empty and the checksums differ, a collision has been found. This is conceptually equivalent to using 64-bit hash indices. Please note that though undetected collisions (i.e., same hash and checksum indices) are extremely unlikely, they remain theoretically possible. However, we were not able to isolate such cases in real-world scenarios.

The simplest collision mitigation technique is a linear search scanning the hash map for an empty entry. This technique is cache-friendly, but areas in the hash map corresponding to collisions become very dense, hence introducing further expensive collisions from unrelated areas (Figure 41-2). Rehashing consists in re-applying a hash function to generate another, noncontiguous index (Figure 41-3). While this impacts cache coherence, it also preserves the distribution uniformity and results in overall increased performance in our tests.

41.2.2 REFINING THE HASH FUNCTION

The spatial hash function just described subdivides the scene into equally sized voxels. In a way similar to bilateral filtering [8], the hash function can be

```
/* Entry allocation                                                    */
hashIndex = H_Wang(P)  mod  hashMapEntryCount;
checksum = H_XORShift(P);
searchSteps = 0;
/* Allow a limited number of searches to avoid deadlocks              */
while searchSteps < MAX_SEARCH_STEPS do
    storedChecksum = atomicCAS(Headers[hashIndex], checksum,
     EMPTY);
    if storedChecksum == EMPTY OR storedChecksum == checksum then
        /* The entry at that index was empty or already exists */
        return hashIndex;
    end
    if storedChecksum != checksum then
        /* Collision found, compute another index                 */
        hashIndex = h(hashIndex)  mod  hashMapEntryCount;
        searchSteps++;
    end
end
return NOT_FOUND;
```

Figure 41-3. *Procedure for allocating hash entries and mitigating collisions.*

refined by including additional criteria, such as the surface normal:

$$H_N(P, \mathbf{n}) = h\left(\lfloor n_z.d_N \rfloor + h\left(\lfloor n_y.d_N \rfloor + h\left(\lfloor n_x.d_N \rfloor + H(P)\right)\right)\right), \tag{41.2}$$

where d_N is a discretization constant for the components of the normal vector. Multiple levels of detail (LOD) can also be stored within the hash map by further including an LOD index to the hash function and adapting the spatial discretization. As shown in Figure 41-4, adding those criteria involves allocating additional hash entries to represent the points visible in the final image. Because this also results in less coherent memory accesses, it is crucial to consider the cost-to-benefits ratio of each added criterion.

41.2.3 STORING INFORMATION IN THE HASH MAP

Each hash map entry contains information, also referred to as its *payload* in this chapter. For example, the payload may be a floating-point RGB irradiance value and a counter representing the number of rays used in the irradiance estimate. Because several threads may modify an entry at once (e.g., tracing more rays), the payload is directly altered using atomic functions

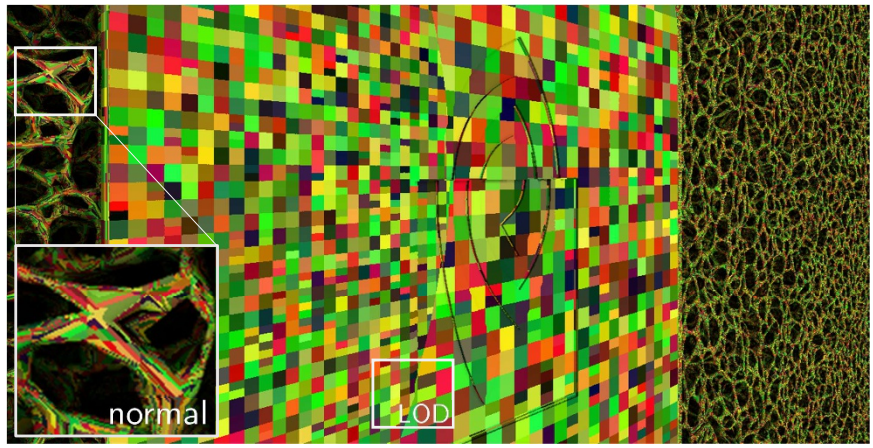


Figure 41-4. *The computation of the hash index (shown in false color) is extended to include normals, hence ensuring points within a hash entry lay on surfaces with similar orientations. The world-space size of the entries are adapted to the viewing distance by adding an LOD index and a discretization size to the hash function as well.*

(Figure 41-5). Note that issuing numerous atomics targeting a single address in global memory may result in a performance loss on graphics hardware. This loss can be particularly significant when the hash map entries cover many pixels in the image, and the payload becomes complex. For further implementation details, we strongly encourage the reader to refer to Binder et al. [2] and Gautron [4].

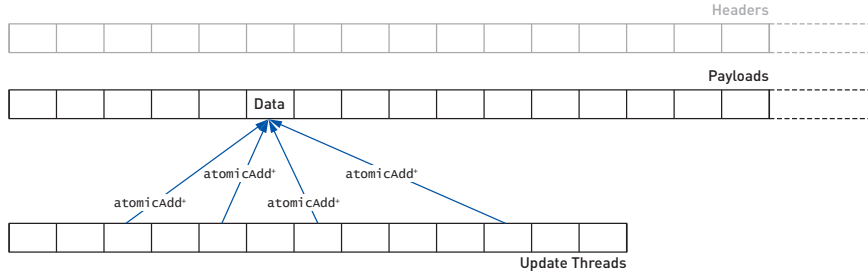


Figure 41-5. *The original spatial hashing algorithm updates hash entries on the fly using atomic functions, which can result in high atomic pressure and data structure restrictions.*

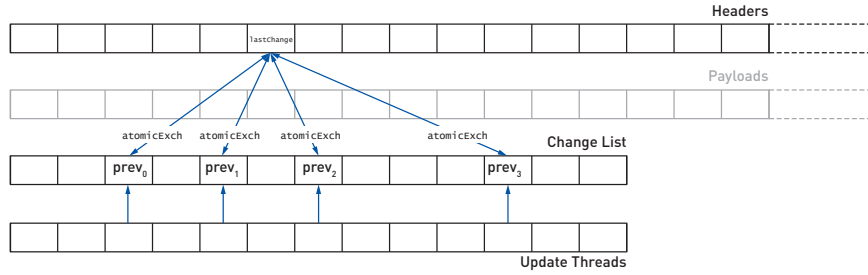


Figure 41-6. *The update step change requests are generated for each update thread, while the hash headers keep track of the index of the last change request touching each entry. Each change request updates that tracker and stores its previous value.*

41.3 COMPLEX DATA STORAGE AND UPDATE

Besides the atomic pressure when using larger payloads, another issue with the direct update is the limitation on the possible data types. For example, though using 16-bit floating-point values would increase cache coherence, neither GLSL for Vulkan nor HLSL provide the corresponding atomic functions. A related issue arises with interdependent components, such as the brightest radiance sample found so far and the index of that sample. A costly mutual exclusion (mutex) would be required to safely update both values.

Instead, we store the changes into a *change list* buffer, where each change is labeled with the index of the hash entry to which it refers. In order to avoid costly sorting operations, we build per-entry change lists on the fly. To this end, we add another unsigned integer `lastChange` to the entry header. When creating the change request *n* in the change list, the algorithm looks up the value `lastChange` of the related entry and atomically exchanges it with the index *n*. The former `lastChange` value is then stored in the `previous` field of the change request. This way, each change request links to the previous change request related to the same entry.

In the example of Figure 41-6, threads 2, 4, 6, and 10 update the same hash entry.[1] Initially we have `lastChange == NO_PRECEDENT`. When thread 2 creates a change request, it atomically exchanges its index with `lastChange`. This sets `lastChange` ← 2 and `previous(2)` ← `NO_PRECEDENT`. Thread 4 performs the same operations, but fetches the updated value of

[1] For clarity we order the operations following the thread numbers. In practice this order is determined by the hardware scheduler.

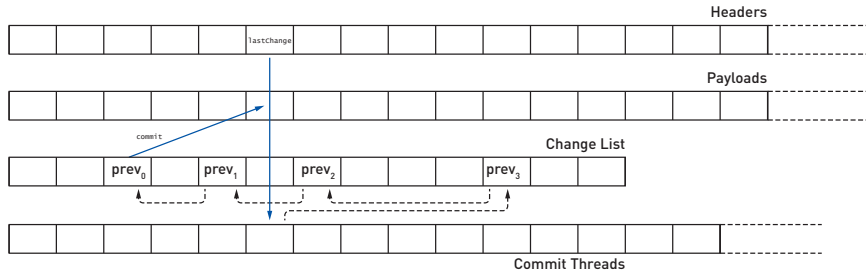


Figure 41-7. *The commit step starts by fetching the index of the last change request for a given hash entry. We then follow the linked list using the indices stored in each of the change requests. After combining the changes, the final update is committed into the hash map.*

`lastChange` == 2. The change request creation yields `lastChange` ← 4 and `previous(4)` ← 2. This is repeated for threads 6 (`lastChange` ← 6, `previous(6)` ← 4) and 10 (`lastChange` ← 10, `previous(10)` ← 6).

At the end of the process, `lastChange` will be the starting point of the commit step, in which the change requests of the modified entries are written into the hash map payloads. This algorithm fetches the change request 10, which links to the previous element in the change list (`previous(10)`== 6), and so on (Figure 41-7). This simple algorithm provides the basis for fast parallel updates of arbitrary hash map payloads. The next section will cover some practical implementation details.

41.4 IMPLEMENTATION

41.4.1 DATA STRUCTURES

The *header* of each hash map entry contains two unsigned 32-bit integers: the collision detection `checksum`, and `lastChange` to link change requests. Because spatial hashing is mainly limited by memory latency, it is crucial to optimize its memory access patterns. In particular, allocating a hash entry potentially requires reading many checksum values to resolve collisions, whereas `lastChange` is only reset once by the thread that allocated the entry. Therefore, the headers are stored in a structure-of-arrays fashion so that all checksums are contiguous. The payloads are stored separately for the same reasons. Their layout, though, depends on their specific data structure and access patterns.

In addition to the hash map, our algorithm uses a *change list* to store the change requests before they are committed. The header of a change request

is an unsigned integer linking to the previous change request for the same entry. This buffer is as large as the largest possible set of simultaneous changes, which is typically the number of pixels in the image. We also keep a buffer representing the unique indices of the hash map entries that have been modified by the change list.

Our algorithm addresses the general case where not all threads may generate a change request, and not all hash entries may be touched at once. We use counters for the change list and the list of touched entries, which are atomically incremented.

41.4.2 HASH ENTRY ALLOCATION

The first step of spatial hashing consists in determining the hash entries corresponding to each shaded point. Once an appropriate entry (i.e., empty or with the same checksum) has been found, we reset the change tracker `lastChange` ← `NO_PRECEDENT`. The change request and touched hash entries counters are also reset to 0.

41.4.3 REQUESTING CHANGES

After generating the information to be stored in the hash map (e.g., computing the incoming radiance value by ray tracing), we reserve an entry in the change list by incrementing its counter. We then fetch the current value of `lastChange` and atomically replace it with the index of the new change request. If this change request is the first one issued for the hash entry (`lastChange == NO_PRECEDENT`), the index of the hash entry is added to the list of touched entries (Figure 41-8). As in the original spatial hashing

```
/* Change generation, typically included in the ray tracing
   kernel                                                    */
foreach Change request r in parallel do
    requestIndex = reserveChangeRequestSlot();
    r.previous = atomicExch(Headers[r.hashEntry].lastChange,
     requestIndex);
    if r.previous == NO_PRECEDENT then
        TouchedList.enqueue(r.hashEntry);
    end
    ChangeList[requestIndex] = r;
end
```

Figure 41-8. *Generation of change requests.*

```
/* Commit the changes to the hash map                                  */
foreach Entry index c in TouchedList in parallel do
    clIndex = Headers[c].lastChange;
    totalChange = emptyContribution();
    while clIndex != NO_PRECEDENT do
        totalChange = merge(totalChange, ChangeList[clIndex]);
        clIndex = Headers[c].lastChange;
    end
    Payloads[c] = merge(Payloads[c], totalChange);
    /* Propagate the combined change to the next coarser LOD  */
    if hasParentLOD(c) then
        parentLODEntry = getParentLODEntryIndex(c);
        parentChangeRequest = createChangeRequest(parentLODEntry,
          totalChange);
        Reapply algorithm with parentChangeRequest;
    end
end
```

Figure 41-9. *Final write of the change list in the hash map, with propagation to coarser LODs.*

scheme, several threads may modify the value of `lastChange` in a given entry simultaneously. However, our change list–based approach uses a single atomic exchange call regardless of the payload size, making it scale better to more complex data.

41.4.4 COMMITTING CHANGE REQUESTS

The header of each touched entry links to the last related change request, which, in turn, indirectly links to all other change requests for that entry. Ideally, the contributions of all the change requests can be merged into one, which is then committed into the hash map. This reduces the global memory traffic by keeping most of the working set in local memory (Figure 41-9). Otherwise, the changes would have to be serially written into the hash map, which introduces an additional cost.

41.4.5 PROPAGATING TO COARSER LODS

Spatial hashing can implicitly store multiresolution data (Figure 41-1) by simply adding a level of detail index into the hash function [2]. Further efficiency can be obtained by propagating information from finer LODs to coarser LODs and by leveraging this approach to locally find a trade-off between noise and spatial accuracy [4]. The payload at the coarser LODs is

obtained as a byproduct of the sampling at the finer LOD: each time new samples are traced, their contributions are also written into the coarser LODs.

Because using a coarse LOD means touching fewer entries that cover many pixels, generating one change request per pixel results in only a few threads processing very long change lists. This, in turn, results in poor utilization on modern graphics hardware running thousands of threads in parallel. However, in most cases change requests can be combined into one. As indicated in the previous section, this merging ability is a key to the efficiency of the update mechanism. Once a change list has been committed to its finer-LOD entry, we create a single, *combined* change request for the same location at a coarser LOD.

This technique requires allocating a second change list where those changes are recorded. It also introduces two additional passes: a cleanup pass clears the change trackers for all touched entries, and a propagation pass builds the list of touched entries from the new change list. The committing pass can then be applied to the newly generated change list. This process can then be repeated to propagate to further, coarser LODs if necessary.

41.5 APPLICATIONS

We applied these techniques for real-time computation of ray traced shadows, applied to ambient occlusion and environment lighting. Our implementation uses the Vulkan API, running on a GeForce RTX 3090 to generate images at resolution 1920×1080.

41.5.1 AMBIENT OCCLUSION

Ambient occlusion (AO) is a very useful approximation of global illumination, where the ambient term is modulated based on the proximity of occluding geometry. In this case the payload of each hash entry contains the number of samples traced so far and the number of actual occlusions found (Figure 41-10).

```
struct Payload contains
    uint16_t occlusions;
    uint16_t sampleCount;
end
```

Figure 41-10. *For ambient occlusion each hash entry fits 32 bits to store occlusion and sample counters.*

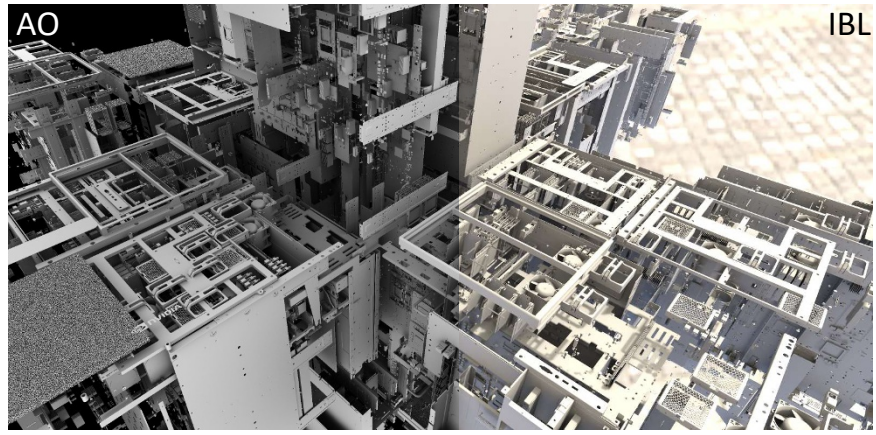


Figure 41-11. *The performance of our update scheme is on par with the atomics-based update using simple data types such as for ambient occlusion (left). It also expands the storage capabilities of the spatial hash map to support more complex payloads such as 16-bit floating-point data (image-based lighting, right).*

Due to its simplicity, this payload structure can be updated directly using a single unsigned integer `atomicAdd` operation, provided the sample count does not exceed 65,535. We compared the direct update with our approach in a CAD scene comprising 304 million triangles (Figure 41-11). All other aspects of the AO computation algorithm being equal, the overall render time per frame for both techniques is equivalent: 7.04 ms for direct updates versus 7.01 ms with our change list–based technique. Though the list management introduces some overhead, this is compensated by the reduced atomic pressure in the LOD propagation. Our algorithm has been integrated in the real-time viewport of the industry-standard Dassault Systèmes 3DEXPERIENCE platform (Figure 41-12).

41.5.2 ENVIRONMENT LIGHTING

This application considers the computation of shadows cast by a spherical environment. The payload of a hash map entry is the accumulated irradiance value at the corresponding location. In order to reduce the memory footprint of the hash map, the payload is defined on 64 bits (Figure 41-13).

Combined with a 64-bit header for each hash entry, and with the typical five million entries in the hash map, the subsequent 128-bit entries result in a memory occupancy of 80 MB. Atomic operations on `f16vec3` types are not supported in GLSL at this time, making our solution the only option for efficient updates.

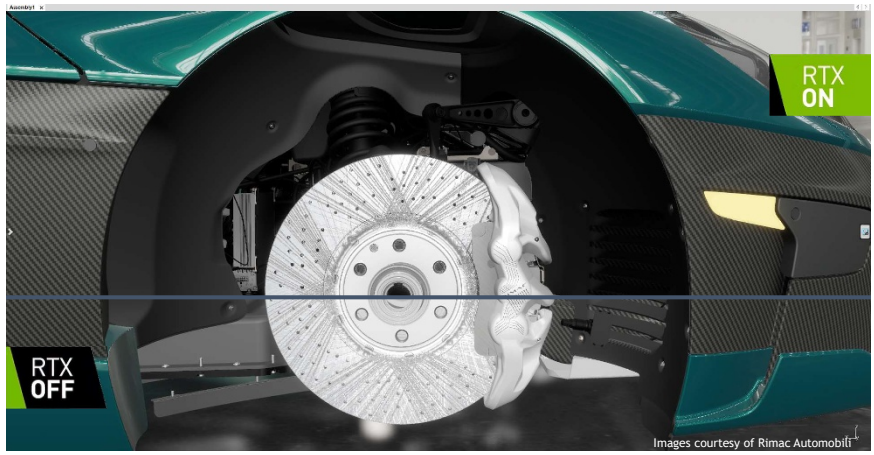


Figure 41-12. *Our algorithm (top) has been integrated within the Dassault Systèmes 3DEXPERIENCE platform as an alternative to SSAO (bottom). The rendered model contains 50 million triangles. (Model courtesy of Rimac Automobili.)*

```
struct Payload contains
    f16vec3 rgb;
    uint16_t sampleCount;
end
```

Figure 41-13. *Environment lighting requires accumulating floating-point values, stored on 16 bits for performance.*

41.6 CONCLUSION

Spatial hashing is a GPU-friendly method for representing sparse data in world space. Despite its efficiency, it originally suffers from limited representation capabilities due to the need of atomic functions to update the contents of the hash map. Using a single temporary variable and a deferred update mechanism, the update technique of this chapter enables dynamic storage and updating of complex data by generating per-entry linked lists. The resulting technique is applied to ray traced ambient occlusion and image-based lighting in massive CAD scenes. This technology has been integrated into the real-time viewport of Dassault Systèmes Catia.

As using hash maps to represent world-space data is relatively new, it carries a strong inspirational value for future work. In particular, the optimization of

the hash function for improved cache coherence holds vast potential for performance optimization. Extensions for nearest-neighbor searches and spatial filtering will also extend the domains in which spatial hashing can be applied.

REFERENCES

[1] Bavoil, L., Sainz, M., and Dimitrov, R. Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 Talks*, 22:1, 2008. DOI: 10.1145/1401032.1401061.

[2] Binder, N., Fricke, S., and Keller, A. Fast path space filtering by jittered spatial hashing. In *ACM SIGGRAPH 2018 Talks*, 71:1–71:2, 2018. DOI: 10.1145/3214745.3214806.

[3] Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A., and Jarosz, W. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions of Graphics (Proceedings of SIGGRAPH)*, 39(4):148:1–148:17, 2020. DOI: 10.1145/3386569.3392481.

[4] Gautron, P. Real-time ray-traced ambient occlusion of complex scenes using spatial hashing. In *ACM SIGGRAPH 2020 Talks*, 5:1–5:2, 2020. DOI: 10.1145/3388767.3407375.

[5] Goral, C. M., Torrance, K. E., Greenberg, D. P., and Battaile, B. Modeling the interaction of light between diffuse surfaces. In *SIGGRAPH '84: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, pages 213–222, 1984. DOI: 10.1145/800031.808601.

[6] Jensen, H. W. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, 2001.

[7] Marsaglia, G. XORShift RNGs. *Journal of Statistical Software*, 008(i14), 2003. https://EconPapers.repec.org/RePEc:jss:jstsof:v:008:i14.

[8] Tomasi, C. and Manduchi, R. Bilateral filtering for gray and color images. In *Sixth International Conference on Computer Vision*, pages 839–846, 1998. DOI: 10.1109/ICCV.1998.710815.

[9] Wang, T. Integer hash function. https://gist.github.com/badboy/6267743, 1997.

[10] Ward, G., Rubinstein, F., and Clear, R. A ray tracing solution for diffuse interreflection. In *SIGGRAPH '88: Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, pages 85–92, 1988. DOI: http://doi.acm.org/10.1145/54852.378490.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 42

EFFICIENT SPECTRAL RENDERING ON THE GPU FOR PREDICTIVE RENDERING

David Murray,[1] Alban Fichet,[1] and Romain Pacanowski[1,2]
[1]Institut d'Optique Graduate School, CNRS
[2]INRIA

ABSTRACT

Current graphic processing units (GPU) in conjunction with specialized APIs open the possibility of interactive path tracing. Spectral rendering is necessary for accurate and predictive light transport simulation, especially to render specific phenomena such as light dispersion. However, it requires larger assets than traditional RGB rendering pipelines. Thanks to the increase of available onboard memory on newer graphic cards, it becomes possible to load larger assets onto the GPU, making spectral rendering feasible. In this chapter, we describe the strengths of spectral rendering and present our approach for implementing a spectral path tracer on the GPU. We also propose solutions to limit the impact on memory when handling finely sampled spectra or large scenes.

42.1 MOTIVATION

Nowadays, computer-generated images are common due to the video game and movie industries' continuous growth. Although the images produced by the entertainment industry look more and more realistic, the process used to render them, named here *tristimulus rendering*,[1] is unsuitable for some domains. For example, the architecture and automotive domains require *predictive rendering* to assess the visual quality of their products before they are put on the production line.

Predictive rendering requires a spectral simulation of light transport. This permits the creation of complex wavelength-dependent effects, which can

[1]A tristimulus renderer computes light transport using only three color channels. The color space (e.g., RGB, HSV, etc.) defining the colors is arbitrarily chosen.

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_42

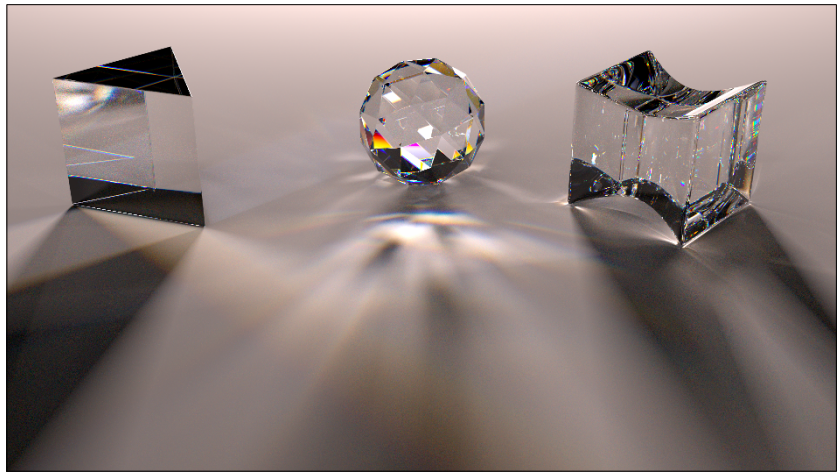


Figure 42-1. *Only a spectral renderer can simulate accurately the richness of color from wavelength-dependent effects, such as light dispersion.*

produce striking colors, such as light dispersion (see Figure 42-1), light polarization, iridescence, opalescence, and even fluorescence effects.

Compared to tristimulus rendering, the usual drawbacks for spectral rendering are its slowness and the memory footprint needed by scene assets. Indeed, assets must be described for each wavelength, which drastically increases the total memory footprint of a scene.

Central processing unit (CPU)–based spectral rendering engines are present in the academic sector (e.g., ART [25], PBRT v3 [23], and Mitsuba [11]) and also in the industry (e.g., Manuka [8]) but are not the norm. On the other hand, most renderers exploiting graphical processing unit (GPU) capabilities are RGB-based (e.g., Iray [14] and Cycles [3]). However, an increasing number of academic spectral renderers now exploit the GPU as well (e.g., Mitsuba 2 [18], PBRT v4 [22], and Malia [5]).

Historically, GPU compute capabilities have grown faster than those of CPUs but are more limited in terms of available memory. However, the recent release of specialized ray tracing hardware for the GPU (e.g., RTCore) and specific APIs (e.g., OptiX [20], DirectX 12 Raytracing, and Vulkan Ray Tracing [24]) provides an opportunity to achieve spectral rendering at interactive frame rates. This opens the possibility to build new pre-visualization tools (e.g., for designers) with spectral rendering capabilities.

In this chapter, we present, through our open source solution Malia [5], how to implement efficient spectral rendering on the GPU. Malia can be used to generate *spectral* images in offline mode (but still using the power of the GPU) or in pre-visualization mode by using progressive rendering and an OpenGL framebuffer to display interactively the current spectral image. These features permit us to showcase step-by-step guidelines for spectral rendering on the GPU while providing appropriate figures to illustrate both its usefulness and its performance.

In Section 42.2, we briefly summarize the theoretical limitations of tristimulus rendering and illustrate why it cannot be used to generate predictive and accurate images. In Sections 42.3 and 42.4, we explain and evaluate various technical solutions implemented in Malia for spectral rendering on the GPU. Then in Section 42.5, we present some results to illustrate what performance can be achieved with these solutions and how to scale the approach with larger assets as well as increased spatial and spectral resolutions of the simulated image sensor. Section 42.6 concludes this chapter with a discussion and outlines potential future work that could improve even further either the accuracy or the efficiency of the proposed solution.

42.2 INTRODUCTION TO SPECTRAL RENDERING

In a tristimulus renderer, all reflectance and illuminance color values are tristimulus. If these values derive from spectral data, then integrating spectra values into tristimulus values is performed prior to the rendering process, comprising its main limitation (see Section 42.2.1). In practice, these values are directly provided in XYZ or RGB color space.

A spectral renderer operates on a per-wavelength basis (see Section 42.2.2), covering the full visible spectrum. Reflectance and illuminance values are directly used as spectral data, thus requiring no transformation. If needed, the final spectrum can be converted to a specific color space using any relevant sensor model (see Section 42.2.3).

42.2.1 LIMITATION OF TRISTIMULUS RENDERING

The integration process, prior to light transport, is the core limitation of a tristimulus renderer. Equation 42.1 illustrates this approximation[2]:

[2]To simplify the notation, we represent only a single bounce. The reflectance spectra or tristimulus values have to be multiplied by each other for each further bounce.

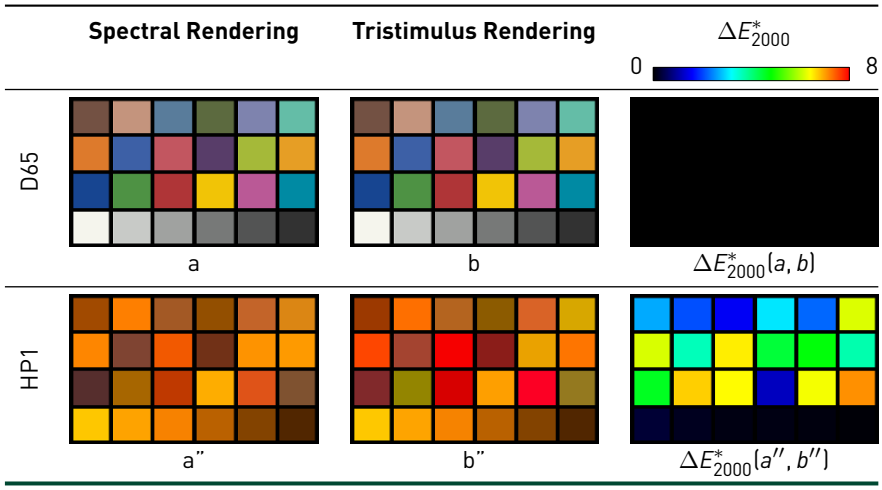


Figure 42-2. *Direct illumination comparisons between spectral and tristimulus renderings for two classical illuminants: D65 (top) and HP1 (bottom). When using spectral rendering, we transform the colors from spectra to RGB at the end. With tristimulus rendering, we transform both reflectance and illuminance values into their tristimulus counterparts prior to the evaluation. Then, we compute the final color by multiplying the illuminance and reflectance colors together. This approximation leads to observable differences as shown by the ΔE^*_{2000} metric.*

$$\underbrace{\int_\Lambda r(\lambda)L_e(\lambda)\bar{c}(\lambda)\mathrm{d}\lambda}_{\text{spectral rendering}} \approx \underbrace{\overbrace{\left(\frac{1}{N_m}\int_\Lambda r(\lambda)L_e^m(\lambda)\bar{c}(\lambda)\mathrm{d}\lambda\right)}^{\text{tristimulus reflectance}} \cdot \overbrace{\left(N_m \frac{\int_\Lambda L_e(\lambda)\bar{c}(\lambda)\mathrm{d}\lambda}{\int_\Lambda L_e^m(\lambda)\bar{c}(\lambda)\mathrm{d}\lambda}\right)}^{\text{tristimulus illuminance}}}_{\text{tristimulus rendering}}, \tag{42.1}$$

where $N_m = \int_\Lambda L_e^m(\lambda)\bar{y}(\lambda)\mathrm{d}\lambda$ is the illuminance normalization factor, $r(\lambda)$ is the spectral reflectance value, $L_e(\lambda)$ is the spectral power distribution (SPD) of the rendering illuminant, $L_e^m(\lambda)$ is the SPD of the illuminant used for measuring reflectance, and $\bar{c}(\lambda)$ is one of the color matching functions $(\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda))$.

The presence of the normalization factor N_m in Equation 42.1 implies that the tristimulus reflectance value will only be correct if $L_e = L_e^m$, that is, only if the exact same illuminant is used for both measuring and rendering. Otherwise, discrepancies occur, as illustrated by Figure 42-2. This is very likely to happen if several different illuminants are used in a scene.

Global illumination increases even more the discrepancies (see [7]) presented earlier. As the tristimulus values are only approximated, sharp power distribution (e.g., the "HP1" illuminant SPD shown in Figure 42-3a) will be



(a) *Illuminant* **(b)** *Tristimulus rendering* **(c)** *Spectral rendering*

Figure 42-3. *Discrepancies induced by a tristimulus renderer are even more prevalent in a global illumination context. This leads to important color and intensity differences, especially with narrow spectra like the one used to light this Cornell box scene (HP1, high-pressure vapor lamp).*

either under- or overestimated. At each new reflection bounce event, a tristimulus renderer thus accumulates error by breaking energy conservation, yielding incorrect results (see Figure 42-3).

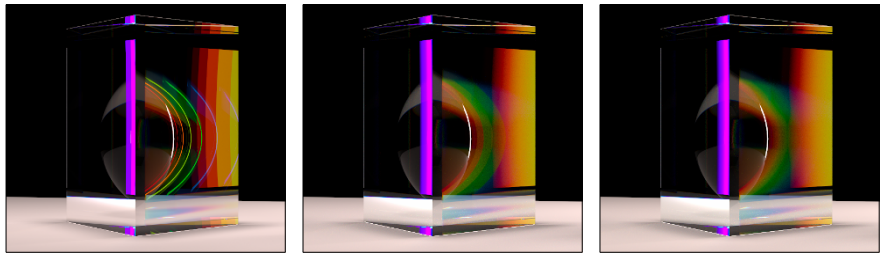
42.2.2 BASIS OF SPECTRAL RENDERING

To implement a spectral renderer, fixed wavelength sampling is the simplest scheme to use. A set of discrete wavelengths is determined at the beginning of the rendering process, and the path tracer will exclusively and exactly use these wavelengths. It offers simple asset management. However, this sampling scheme produces, most of the time, severe visual color artifacts (e.g., spectral aliasing). In particular when dealing with wavelength-dependent paths (i.e., refracted rays), only a discrete set of the possible paths is explored (see Figure 42-4a).

To avoid spectral aliasing, another dimension needs to be taken into account in the Monte Carlo integration process: the spectral domain (see Evans and McCool [6]). In this chapter, the spectral domain is fully covered by jittering the processed wavelength within an interval of two fixed wavelengths (see Figure 42-4b). We will refer to this interval as a *spectral bin*. To further reduce aliasing, the resulting value can be distributed to the adjacent spectral bins with any desired kernel for filtering (i.e., with a linear interpolation, see Figure 42-4c).

42.2.3 OUTPUT OF A SPECTRAL RENDERER

A spectral renderer can output a spectral image, a color image, or both. The choice mainly depends on the application. *Color images* require that the



(a) *Rendering with a discrete set of wavelengths*

(b) *Sampling wavelength at each new sample; accumulation in nearest bins*

(c) *Sampling wavelength at each new sample; accumulation distributed in neighboring bins*

Figure 42-4. *This scene shows a prism made of a dispersive glass: it has a wavelength-dependent index of refraction (later detailed in Section 42.4). (a) A discrete set of wavelengths are used. This produces significant spectral banding artifacts because only a subset of paths is explored. (b) We jitter the wavelength within each bin for each sample and then accumulate the resulting radiance in the nearest bins. There is still some banding because the transitions between sampled bins are visible in the sphere reflection. (c) We use a different kernel to accumulate results in neighboring bins, thus further decreasing the hard transitions between bins.*

sample accumulation is performed in a specific color space (XYZ, RGB, CMYK, etc.). Samples are thus converted prior to the accumulation. The main benefit of generating directly a color image is that the result will be the most accurate possible for this specific color space. However, the main downside of this format is that it is hardly transposable to another color space.

On the other hand, a *spectral image* remains "color space agnostic." As it contains the spectral power distribution, it can be converted to any color space (e.g., a display device or a printer). The downside of a spectral image is the additional storage cost as it requires one image layer per spectral bin. Note that an interactive renderer may generate, internally, a spectral image while displaying a color image by performing on-the-fly conversion. An important advantage of spectral images is that they store physical units, which is especially desirable for predictive rendering. For this reason, they also can be reused as the input to a spectral renderer.

One must be aware that there are two types of spectral images: *emissive* (storing energy $\in [0, \infty)$) and *reflective* (storing reflectance or attenuation values $\in [0, 1]$) images. When converting a spectral image to a color space, the process is not exactly the same depending on its type, as detailed in Figure 42-5.

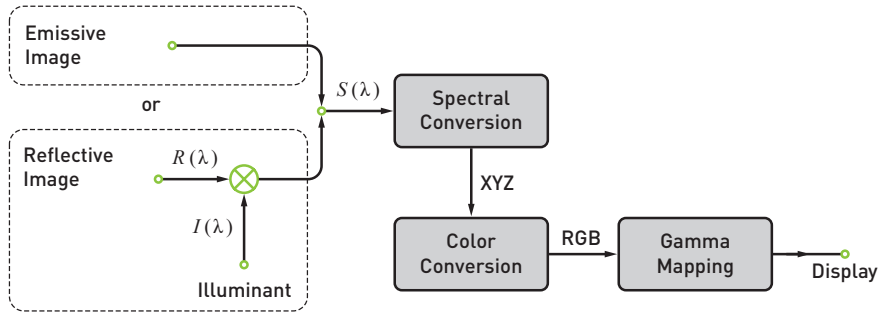


Figure 42-5. *Typical conversion pipeline for spectral images. Reflective spectral images have to be multiplied by an illuminant SPD prior to the spectral to tristimulus conversion. This ensures getting a preview of the reflectance given a specific illumination condition.*

In our renderer, we use spectral outputs: we want to retain the full radiometric quantities while having the ability to simulate the impact on different sensors capturing the same scene after the rendering process (e.g., a sensor from Ximea [4]).

In summary, spectral rendering is necessary when the following occur:

- Targeting color-accurate results regardless of the illuminant used in the scene.
- Using the resulting image on various and unknown display devices (i.e., need for "color space agnostic" images).
- Rendering specific visual effects implying wavelength considerations (e.g., dispersion, polarization, etc.).

42.3 SPECTRAL RENDERING ON THE GPU

Limited memory is the main challenge when using GPUs for spectral rendering: spectral assets are larger than their tristimulus counterparts. Often, all assets cannot fit at their full spectral resolution on video RAM (VRAM), and we must resort to frequent host-device asset transfers. A balance between VRAM usage, host-device transfers, and computation time is crucial for efficient rendering, all the more for predictive interactive renderers, as discussed later on in Section 42.5. Our implementation choices are guided by these current hardware limitations.

With these limitations in mind, we use a single wavelength approach (one wavelength per ray) to introduce the different strategies that can be

considered when implementing a spectral renderer. Then, using one of the described strategies, we show how *multiplexing* (multiple wavelengths per ray) increases the efficiency of a spectral renderer on a GPU. In particular, we present how we handle efficient wavelength sampling in this context, as this can be a pitfall when converting a tristimulus renderer to a spectral one.

42.3.1 SPECTRAL SAMPLING ON GPU FOR SINGLE WAVELENGTH RENDERING

The main difficulty when going from a CPU-based to a GPU-based spectral path tracer is sampling the spectral domain correctly. This process can no longer be efficiently performed on a per-ray basis due to the necessary dispatch on the GPU. It requires all assets at their full spectral resolution to be resident in VRAM if the scene can fit; otherwise, for each new sampled wavelength, all assets must be updated on GPU memory.

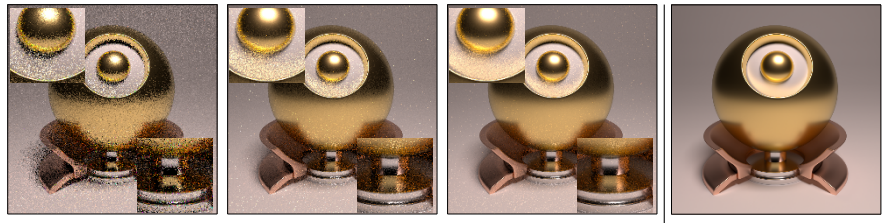
There are three main solutions for implementing single wavelength rendering on the GPU:

1. *Per-wavelength asset upload:* A random wavelength is sampled on the CPU, and all assets are uploaded to the GPU memory for this specific wavelength. This solution is memory-friendly but induces costly CPU-GPU exchanges (one per processed wavelength). Although being inefficient due to the important CPU-GPU transfers, this method provides the reference solution because it does not introduce any bias.

2. *Fixed wavelength asset upload:* A fixed set of spectral bins are processed sequentially. This requires only a subset of the asset spectral data to be uploaded in VRAM. Each bin is centered around one of the user-requested wavelengths. Only this central wavelength is used (fixed sampling). This solution may cause spectral aliasing, as illustrated in Section 42.2.2 and Figure 42-4a.

3. *Wavelength boundary asset upload:* A set of spectral bins is processed sequentially. Assets are uploaded on VRAM for the boundaries of each bin (see Figure 42-7). Sampling a new wavelength is done on the GPU by sampling an offset $\xi \in [0, 1]$. This offset is used to jitter the currently processed wavelength within the bin's boundaries. The assets' values are then *linearly* interpolated on the GPU using this offset.

   This method allows continuous wavelength sampling while limiting the number of CPU-GPU transfers. Note that there is a potential bias when using tabulated data with a resolution greater than the number of bins.



(a) *Single wavelength: 1 wavelength per ray 64 rays per pixel Time: 0.252 s*

(b) *Multiplexed: 32 wavelengths per ray 16 rays per pixel Time: 0.217 s*

(c) *Multiplexed: 32 wavelengths per ray 64 rays per pixel Time: 0.783 s*

(d) *Converged reference*

Figure 42-6. *Convergence comparisons between (a) single wavelength and (b,c) multiplexed approaches. The convergence is faster with multiplexing—(c) for the same number of rays per pixel and (b) for the same time—compared to the single wavelength rendering. The scene is composed of only metallic materials (i.e., conductors). All rendered images (*512×512 *pixels) are generated with an NVIDIA RTX 3070 with 32 wavelengths evenly distributed between 380 and 750 nm.*

In the following sections, we describe more extensively the boundary asset management for multiplexed rendering. The reasoning remains the same for single wavelength rendering, each bin being processed one at a time.

42.3.2 WAVELENGTH MULTIPLEXING

The efficiency of a spectral renderer can be greatly improved by using multiplexing: processing a single ray that carries multiple wavelengths (i.e., a spectrum) instead of a single wavelength. The benefits of this approach are twofold. First, it reduces the number of costly ray-intersection computations, and second, it also reduces the number of BRDF importance sampling and evaluation events.

Implementation-wise, adding simple multiplexing support to a single wavelength path tracer is straightforward. Instead of processing a single wavelength, a ray carries multiple wavelengths at once.[3] All computations are vectorized to handle all wavelengths. When propagation is wavelength-independent (e.g., no refractive materials), this approach improves the convergence significantly, as illustrated in Figure 42-6, and also reduces color noise.

This solution is GPU-friendly and can be interpreted as a binary version of the Hero Wavelength Spectral Sampling (HWSS) proposed by Wilkie et al. [26].

[3] RGB rendering may be, in a way, considered as a special case of multiplexing.

HWSS also computes and stores the probabilities that the current path is valid for the wavelengths it conveys. This allows efficient handling of media with wavelength-dependent scattering (e.g., participating media or fluorescent elements).

42.3.3 ENFORCING CONTINUOUS SPECTRAL SAMPLING WITH MULTIPLEXING

Given the GPU-oriented approach, we have additional constraints regarding asset management. To avoid costly CPU-GPU transfers, instead of uploading the assets for the sampled wavelength, we upload the assets for wavelengths at the upper and lower boundaries of each requested bin. Then, the random wavelength sampling is done on the GPU within each bin, and the assets are interpolated on the fly using the wavelength offset between the lower and upper bounds (see Figure 42-7).

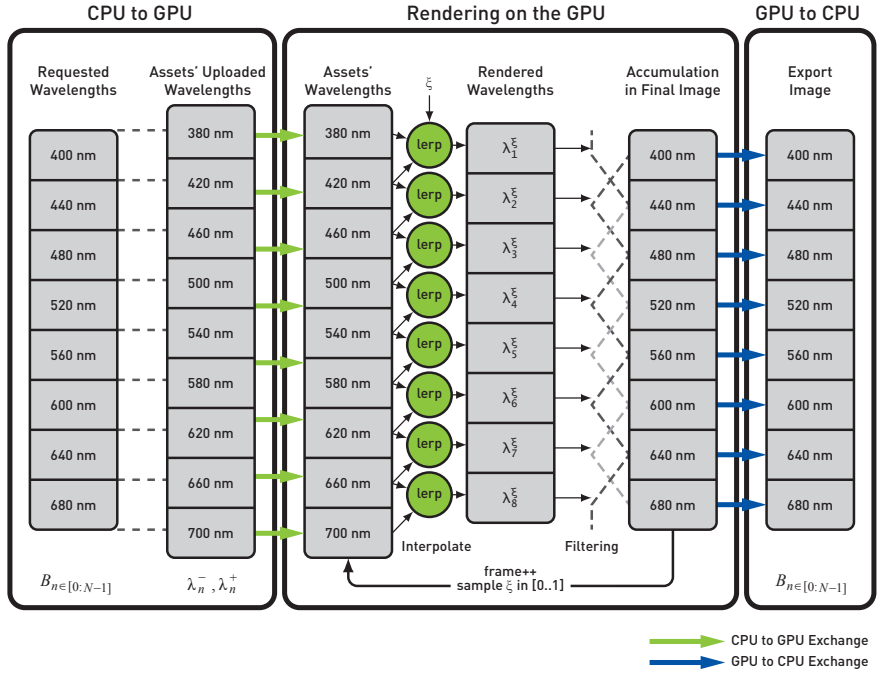


Figure 42-7. *An overview of the proposed pipeline. First, we determine the boundary of each spectral bin. Then, the assets are uploaded to the GPU for the bin boundaries. At each sample, a random offset ξ allows jittering the wavelength in the interval $[\lambda^-, \lambda^+]$. Each tabulated asset is evaluated using a linear interpolation between the two adjacent boundaries. Then, the result is accumulated to a spectral buffer, representing the requested wavelengths, and filtered to mitigate further spectral banding. Finally, the accumulation buffer is transferred to the CPU on request for saving the spectral image.*

This may introduce a small bias for tabulated assets but permits efficient rendering, by sparing a tremendous amount of data transfer. Note that such an approach remains completely unbiased with analytical spectra, or when the spectral resolution of the assets is lower than half the spectral resolution of the final image (Nyquist–Shannon sampling theorem).

UPLOADING ASSETS ON THE GPU

For a given set of N continuous bins $B_{n\in[0:N-1]}$ centered on the requested wavelengths λ_{B_n}, we upload the assets for the lower, λ_n^-, and upper, λ_n^+, boundaries of each bin (see Figure 42-7). The uploaded asset values at λ_n^- and λ_n^+ take the form of an average value, A_{λ_i}, to ensure energy conservation:

$$A_{\lambda_i} = \frac{1}{\Delta} \cdot \int_{\lambda_i - \frac{\Delta}{2}}^{\lambda_i + \frac{\Delta}{2}} A(\lambda)\, \mathrm{d}\lambda\,, \tag{42.2}$$

where λ_i is the wavelength at the boundary $\lambda_i = \lambda_n^-$ or $\lambda_i = \lambda_n^+$, $A(\lambda)$ is the value of the asset at λ, and Δ is the spectral bandwidth of the bin (i.e., $\Delta = \lambda_n^+ - \lambda_n^-$)

In the particular case of adjacent bins, we upload $N + 1$ spectral elements for each asset because the upper bound λ_{n-1}^+ of B_{n-1} matches the lower bound λ_n^- of B_n:

$$\lambda_n^+ = \lambda_{n+1}^- = \lambda_{B_n} - \frac{\lambda_{B_n+1} - \lambda_{B_n}}{2}\,. \tag{42.3}$$

At this point, we have access to $N + 1$ spectral values for each asset. However, we still propagate N wavelengths with each ray for the whole rendering process. For each of these wavelengths, we interpolate the asset value using two values from the $N + 1$ values stored on the GPU.

Note that this does not hold when using nonconsecutive bins, as in Section 42.5.3.

WAVELENGTH SELECTION

For each new ray, we sample an offset $\xi \in [0, 1]$ between the lower, λ_n^-, and upper, λ_n^+, boundaries of each spectral bin carried by this ray. Therefore, each new ray explores a different subset of wavelengths.

For a given random offset ξ, the corresponding wavelength λ_n^ξ for the bin B_n is bounded between λ_n^- and λ_n^+ and computed as follows:

$$\lambda_n^\xi = \lambda_n^- + \xi \cdot (\lambda_n^+ - \lambda_n^-)\,. \tag{42.4}$$

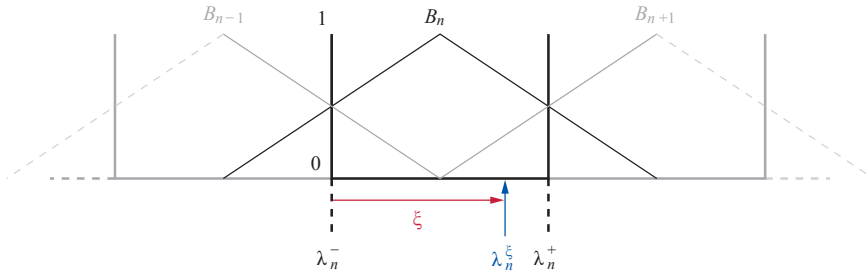


Figure 42-8. *After rendering a sample, the energy is accumulated in neighboring bins. We use triangular kernels to accumulate offset samples and attenuate potential spectral banding.*

ACCUMULATION IN SPECTRAL BINS

Once the radiance values L_e are computed for each wavelength λ_n^ξ carried by the ray, we accumulate the radiance into the final image. To mitigate potential banding artifacts, we use a triangular kernel to distribute the ray energy in neighboring bins (see Figure 42-8).

For the nth bin, the weights w are computed as follows:

$$w_{n-1} = \max(0, 0.5 - \xi), \quad w_n = 1.0 - \left|0.5 - \xi\right|, \quad w_{n+1} = \max(0, -0.5 + \xi). \tag{42.5}$$

Then, we can accumulate the resulting radiance $L_e(\lambda_n^\xi)$ in bins B_{n-1}, B_n, and B_{n+1}:

$$B_{n-1} = w_{n-1} \cdot L_e(\lambda_n^\xi), \quad B_n = w_n \cdot L_e(\lambda_n^\xi), \quad B_{n+1} = w_{n+1} \cdot L_e(\lambda_n^\xi). \tag{42.6}$$

42.3.4 SUMMARY

In this section, we first presented the different strategies for spectral sampling with their strengths and weaknesses with respect to the quality and GPU rendering constraints. We explained how to handle spectral multiplexing to improve a spectral renderer's performance by minimizing CPU to GPU transfers. Finally, we showed how to reduce spectral banding artifacts by using filtering during the accumulation step.

Our method can easily be extended to fully support HWSS with the addition of a path probability for each computed wavelength, increasing computation and memory usage. More complex filters can also be used to further reduce the spectral banding artifacts.

42.4 MULTIPLEXING WITH SEMITRANSPARENT MATERIALS

The main limitation of multiplexing appears when the scene contains semitransparent materials. In fact, this limitation arises with any material or medium where the geometrical path taken by the light depends on the considered wavelength. As discussed by Wilkie et al. [26], it forces the multiplexed renderer to operate in single wavelength mode.

In this section, we present a method to improve the efficiency of a multiplexed renderer when a ray hits a semitransparent material as such materials are the most common wavelength-dependent materials. First, we recall why semitransparent materials are not straightforward to handle and illustrate the limitation with multiplexing. Then, we present a solution to overcome partially this limitation and improve the efficiency of multiplexed renderers.

42.4.1 LIMITATION WITH SEMITRANSPARENT MATERIALS

At the wavelength scale, each medium's optical behavior can be characterized by its index of refraction (IOR). The IOR is a dimensionless number that represents, per wavelength, how fast the light speed is affected by the medium. The IOR is a complex number defined by

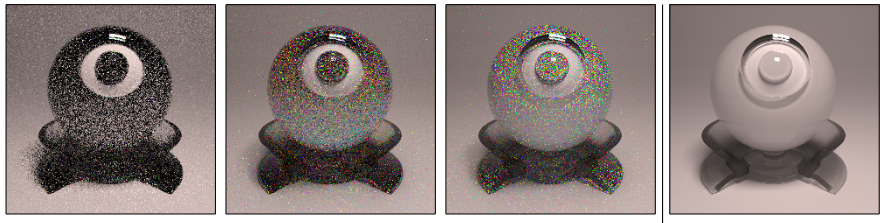
$$n(\lambda) = \eta(\lambda) + j\kappa(\lambda)\,, \tag{42.7}$$

where $\kappa(\lambda)$ represents how much a medium absorbs the light. When $\kappa \gg \eta$, the material is said to be a conductor (e.g., metals). Conductors mostly reflect or absorb light and barely transmit it, whereas dielectrics ($\eta \gg \kappa$, e.g., glass) reflect, absorb, and transmit light.[4] When a light wave changes medium, its geometrical path will be modified according to the Snell–Descartes law:

$$n_i(\lambda)\,\sin\theta_i = n_t(\lambda)\,\sin\theta_t \leftrightarrow \sin\theta_t = \frac{n_i(\lambda)}{n_t(\lambda)}\,\sin\theta_i\,, \tag{42.8}$$

where θ_i is the incident angle and θ_t is the transmitted angle, which defines the new direction of the wave. For a given θ_i, the resulting θ_t will depend indirectly on the wavelength (through the IOR). This dependence implies that only one couple (θ_i,θ_t) is valid for a specific wavelength. Because a multiplexed approach is valid as long as the light path is valid for all the wavelengths carried by the ray, *it cannot be used directly when handling such transmission events*.

[4] For more information on the IOR definition and its various implications in light behavior, we refer the interested reader to the book *Optics* by Eugene Hecht [10].



(a) *Single wavelength: 1 wavelength per ray 64 rays per pixel Time: 0.317 s*

(b) *Multiplexed: 32 wavelengths per ray 16 rays per pixel Time: 0.283 s*

(c) *Multiplexed: 32 wavelengths per ray 64 rays per pixel Time: 1.022 s*

(d) *Reference (single wavelength approach, precision of 1 nm)*

Figure 42-9. *Convergence comparisons of a scene with three semitransparent materials, the IORs of which are constant per wavelength. In this scene, the many decimation events necessary with (b,c) the multiplexed approach introduce color noise. However, multiplexing still offers a gain compared to (a) the single wavelength approach at equal rendering time. All images (*512×512 *pixels) are rendered on an NVIDIA RTX 3070 with 32 wavelengths evenly distributed between 380 and 750 nm.*

When crossing a transparent material, the rendering approach must fall back into a single wavelength propagation (i.e., the incoming spectrum is decimated to a single wavelength [6]). When the wavelength selection is done randomly, color noise reappears for this material type, as illustrated in Figure 42-9. It is important to note that this selection can be done on the fly, so that a light path without any transmission remains multiplexed. This still allows the multiplexed approach to perform well.

An option to overcome the remaining color noise problem could be to split the multiplexed ray into several single wavelength rays (each of them with its own geometrical direction [6]). However, this kind of branching is hardly recommended with CPU-based path tracers due to potential exponential workload. It may be even worse with a GPU-based engine where a compute unit is more easily overloaded than a CPU core, thus significantly increasing the rendering time.

42.4.2 IMPORTANCE SAMPLING FOR THE PROPAGATED WAVELENGTH

For (even slightly) colored dielectrics, a more GPU-friendly approach is to use importance sampling when choosing the wavelength that we want to keep, to make the best of this bad situation. Theoretically, the colored aspect of the medium comes from the IOR (both the real and imaginary parts) as stated by the Fresnel equations on the surface, but also by the type of particles (and their concentration) composing the medium. The most common way to

compute the spectral attenuation induced by the particles of the medium is to apply the Bouguer–Beer–Lambert law[5] $T_{\text{BBL}}(\lambda)$, which quantifies the transmittance rate:

$$T_{\text{BBL}}(\lambda, l) = e^{-\sigma(\lambda)l}, \tag{42.9}$$

where l is the distance (in meters) traveled by the light inside the medium and $\sigma(\lambda)$ is the absorption coefficient, related to the IOR by

$$\sigma(\lambda) = \frac{4\pi\kappa(\lambda)}{\lambda}. \tag{42.10}$$

To select the transmitted wavelength, we propose to build a 1D cumulative distribution function (CDF) for a fixed distance l, from the transmittance function $T(\lambda)$ defined as follows:

$$T(\lambda) = T_{\text{BBL}}(\lambda, 1) \cdot T_{\text{user}}(\lambda), \tag{42.11}$$

where $T_{\text{user}}(\lambda)$ is an additional and ad hoc attenuation rate allowing the user to choose to control which wavelengths are absorbed by the medium. The main benefit of building a CDF from Equation 42.11 is that it only depends on the wavelength and remains small in terms of memory footprint.

The complete importance sampling procedure is the following:

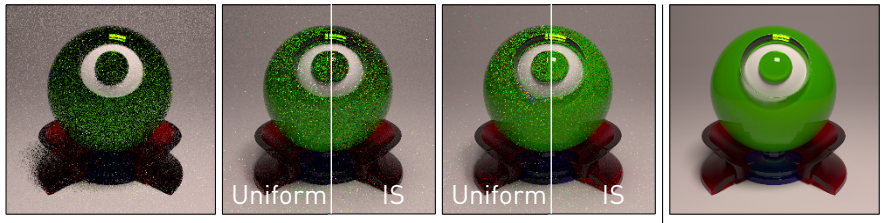
1. Build a CDF from $T(\lambda)$ for each material.[6] This is done on the CPU as a preprocess.
2. When a ray encounters a transparent material, sample a wavelength by inverting on the fly (on the GPU) the previously computed CDF: $\lambda = CDF^{-1}(x \in [0, 1])$.
3. Apply the corresponding probability density function, $pdf(\lambda) = T(\lambda)/\int T(\lambda)$.
4. Propagate the transmitted ray using only the sampled λ.

Note that for a GPU-based path tracer, only the first operation is performed on the CPU while loading the scene. The remaining operations are part of the path tracer execution on the GPU, hence they do not induce any extra CPU-GPU exchanges.

When the transmittance does not present any significant variation, one would still prefer to directly use uniform sampling because, in that case, importance sampling will be close to uniform sampling.

[5] We neglect the case of thin layers, which produces interference. In that case, the Bouguer–Beer–Lambert law is no longer valid (see Mayerhöfer et al. [16]).

[6] In our implementation, we use a 1 nm resolution for λ to compute the CDF and an arbitrary constant distance.



(a) *Single wavelength: 1 wavelength per ray 64 rays per pixel Time: 0.308 s*

(b) *Multiplexed: 32 wavelengths per ray 16 rays per pixel Time: 0.281/0.280 s*

(c) *Multiplexed: 32 wavelengths per ray 64 rays per pixel Time: 1.03/1.02 s*

(d) *Reference (single wavelength approach, precision of 1 nm).*

Figure 42-10. *Comparison of the different wavelength importance sampling (IS) procedures for a scene with different semitransparent colored materials. For this scene, the convergence rate is faster when using importance sampling (b,c, right side) rather than uniform sampling (b,c, left side) to select the propagated wavelength. Note that uniform sampling is still better than using (a) a single wavelength approach thanks to multiplexed reflections. All images (*512 $\times$ 512 *pixels) are rendered on an NVIDIA RTX 3070 with 32 wavelengths evenly distributed between 380 and 750 nm.*

Figure 42-10 illustrates how our importance sampling approach is useful with another difficult scene containing mostly colored dielectrics. At equal time, using importance sampling for the propagated wavelength greatly reduces the variance. Once again, the multiplexed approach is still better than the single wavelength approach.

As stated previously, the most accurate approach for wavelength importance sampling should also consider the incoming ray direction, the distance to the exit point, and the Fresnel transmission term. However, this aspect will be the subject of further study to assert if the gain in performance is worth the additional computational cost and memory usage.

42.4.3 SUMMARY

In this section, we have shown how multiplexed rendering can increase performance, even when dealing with strictly path-dependent materials such as dielectrics. We proposed a method to reduce the impact of the dielectric on multiplexed performance by using an importance sampling procedure to select the appropriate wavelength. This drastically reduces color noise and improves even more the efficiency. This technique is also well suited for GPU renderers with minimal CPU-GPU exchanges. Only a simple precomputation is performed, per transparent material, and only once at the scene loading stage.

This technique can also be used in conjunction with HWSS. Instead of selecting the Hero wavelength at the initial ray launching step, it may be done at the first scattering event. If this event is a dielectric intersection, this would improve HWSS efficiency where the renderer has to operate in single wavelength mode.

42.5 A STEP TOWARD REAL-TIME PERFORMANCE

Real-time rendering with Monte Carlo path tracing[7] is currently hardly possible on a home desktop computer with a tristimulus approach. If we focus on *predictive* and *accurate* rendering of complex materials, where spectral rendering is almost mandatory, it is even more difficult to achieve such frame rates, due to memory limitation and heavier workload. Furthermore, there is an additional, on-the-fly, spectral-to-RGB conversion to display the image, yet another post-process treatment.

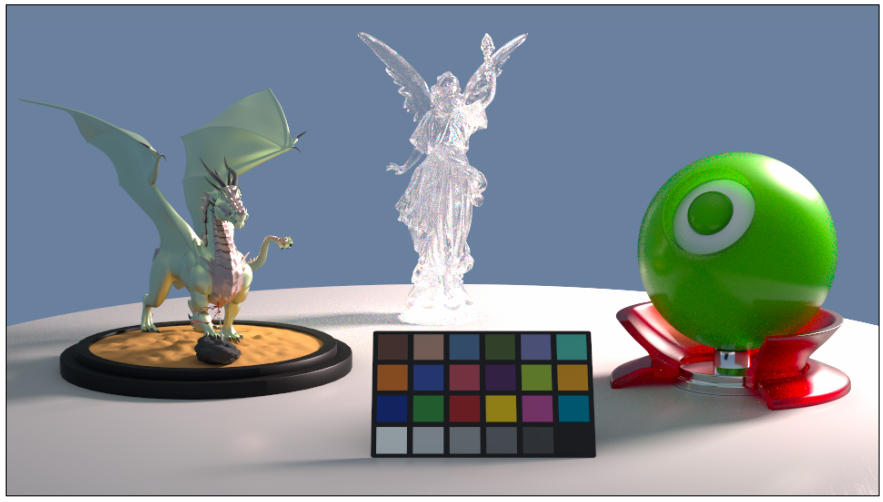
However, we demonstrate in this section that with a limited number of wavelengths and a fully multiplexed approach, one can hope to reach interactive frame rates with a spectrally accurate enough result. We also illustrate how some simple yet useful tools can be implemented to improve performance when one has limited compute capabilities or limited memory.

42.5.1 INTERACTIVE SPECTRAL RENDERING WITH MULTIPLEXING

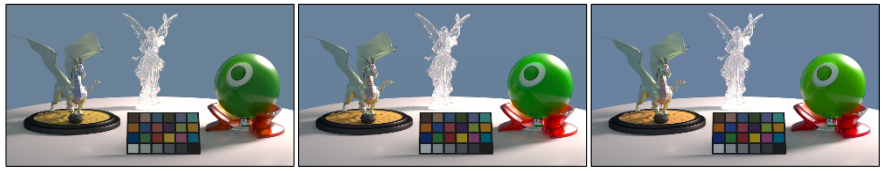
Multiplexing is a key feature when one wants efficient progressive rendering for an interactive purpose, or to aim at real-time spectral path tracing. To support this claim, we provide some results obtained with our solution Malia [5] in which all the aforementioned features are implemented. It is an academic-oriented path tracer (currently OptiX-based [20]), designed to offer predictive and accurate rendering with full support of measured and tabulated materials (e.g., coming from a BRDF acquisition process). As such, our solution is probably not optimized as well as a production one. We still wish to provide hints on the performance that can be achieved with our approach. All results are presented with respect to their RGB equivalent (using the same pipeline except for the asset management) to appreciate the relative performance.

Figure 42-11 illustrates that 16 wavelengths may be sufficient with many "everyday real-world" materials and even some complex ones (some spectra are highlighted in Figure 42-13). Sixteen wavelengths will be more than

[7] At 30 to 60 converged frames per second, with path length up to 10 bounces.



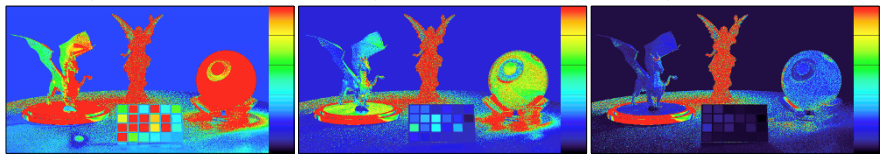
(a) *Reference image with 64 bins*



(b) *Rendering with 8 bins, 125.3 rays/pixel/second*

(c) *Rendering with 16 bins, 87.7 rays/pixel/second*

(d) *Rendering with 32 bins, 49.7 rays/pixel/second*



(e) $\Delta E^*_{2000}(a,b)$, *scale from 0 to 10*

(f) $\Delta E^*_{2000}(a,c)$, *scale from 0 to 10*

(g) $\Delta E^*_{2000}(a,d)$, *scale from 0 to 10*

Figure 42-11. *Comparisons (using CIE ΔE^*_{2000}) of the impact of the number of processed spectral bins, with converged rendering. Common materials (e.g., the MacBeth chart and the table) are depicted in a satisfying way with as few as 16 bins, and nearly perfectly depicted with 32. However, for complex materials (e.g., Lucy, see Figure 42-13 for a peak at the material), 16 bins may not be sufficient. In all cases, only eight wavelengths is not enough for such a scene. For reference, the same scene in RGB requires 6 ms to shoot one ray for each pixel (960 × 540 pixels), that is, 167.63 rays per pixel per second. We can see that the rendering time is nearly linear with the number of processed wavelengths. Performance data are given as the number of rays per pixel per second, the higher the better. All rendered images are generated with an NVIDIA RTX 3070 with wavelengths evenly distributed between 380 and 750 nm.*

enough for most applications that do not use specific and complex spectral materials (e.g., fluorescent materials and interferential materials, such as diffraction gratings or photonic crystals, see Joannopoulos et al. [13]). Note that in this claim, we are merely considering that the image is observed by a human eye. It may not stand in the case of a digital sensor with a fine spectral resolution.

With this in mind, Figure 42-11 also gives some rendering times as rays per pixel per second. We can see that, with our approach, simple scenes like the table (and the probe, see Figure 42-6) are fast to render with up to 32 rays per pixel, producing an image that is converged enough to be efficiently denoised. Targeting approximately ten true frames per second may be realistic as long as a denoising post-process is applied and is performant enough (see Section 42.6).

42.5.2 INTERACTIVITY: SPATIAL SUBDIVISION

If the aforementioned techniques are not sufficient to achieve interactivity, focusing the workload on a subset of pixels is a common approach for real-time RGB renderers. Spectral rendering is no different in this case.

Within Malia [5], as an example, this process is done in two steps. First, we define small square tiles inside the OptiX buffer (typically of four or nine pixels). Then, instead of processing one ray per pixel, we process one ray per tile. The ray spatial origin is jittered inside the tile, and its final value is stored in the corresponding pixel. Finally, when synchronizing the OptiX buffer with an OpenGL framebuffer for display, the framebuffer is filled by splatting the tiles with a simple weighted average (see Figure 42-12), where the weights are the radiance values integrated over the spectrum (that is, the Y channel of the XYZ intermediate value). Note that the content of the OptiX buffer (which may be saved) is still filled using its full original spatial resolution thanks to the jittering within the tiles.

Using square tiles with an adaptive size and splatting the intermediate result on the whole tile is an easy way to reduce the workload per frame. However, more advanced and efficient approaches can easily be used in a spectral context as long as they only operate on the spatial distribution of primary rays (e.g., path guiding by Guo et al. [9] reduces the workload by focusing on important pixels, whereas load balancing by Antwerpen et al. [1] operates in the context of multiple GPUs).

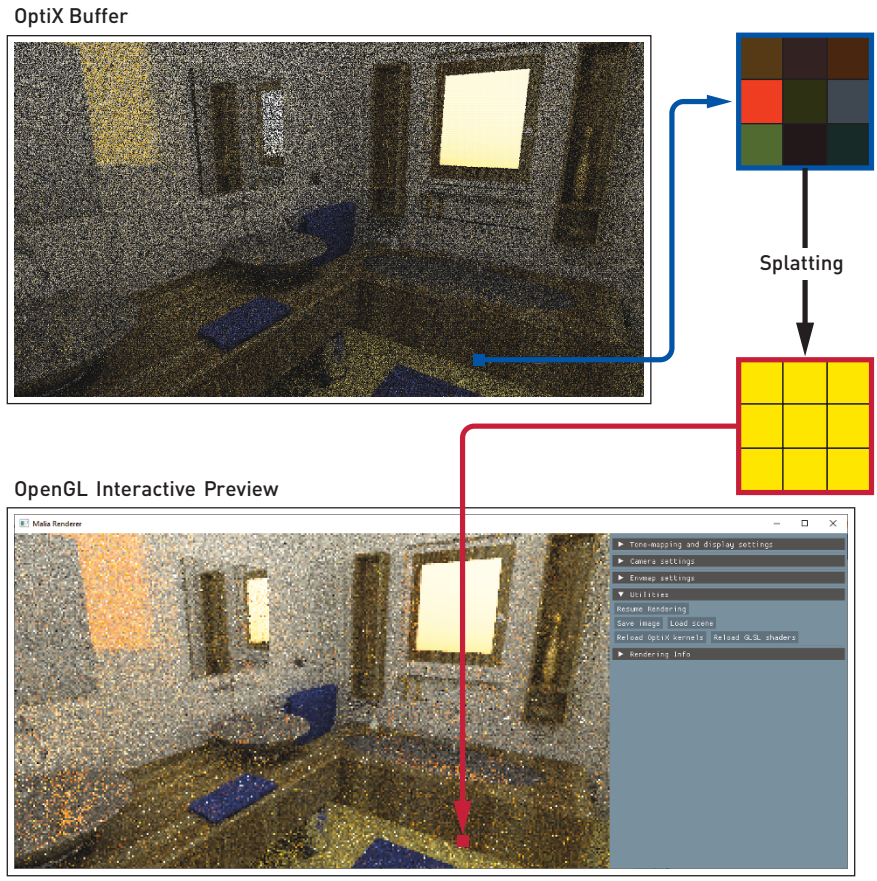


Figure 42-12. *Splatting process. To increase the interactivity, we degrade the spatial resolution by alternatively rendering one of the pixels within a given tile. The OptiX buffer remains at the full resolution to allow exporting the requested resolution.*

Although spatial subdivision is useful to reduce the per-frame workload, and therefore to increase the interactivity, the overall efficiency is significantly reduced due to the additional host calls and to the per-frame post-processing.

42.5.3 LIMITED MEMORY: MULTIPLE SPECTRAL PASSES

Memory can limit the amount of spectral bins that can be handled at once. Moreover, the size of a spectral bin impacts interactivity: a single frame takes longer to render when a higher number of spectral samples is requested. To leverage these issues, two options are possible:

1. *Continuous bin rendering:* The bins are rendered in order. We use multiple rendering passes to cover the whole spectral domain. This method has a minimal memory footprint: the upper bound of a bin and the lower bound of the next one share the same value. So, for N_{bins} rendered at once, we have to upload $N_{\text{bins}} + 1$ assets, as shown in Section 42.3. On the other hand, this method is not optimal for getting a fast preview: color previewing is only possible when all the passes are done.

2. *Interleaved progressive bin rendering:* To improve interactivity, one may want improved color rendering even with the first spectral passes. With an interleaved progressive bin rendering, the rendered bins are equally distributed over the whole spectral domain. So, color accuracy is increased at each pass. However, in this case, two bins do not share a common boundary anymore. So, for N_{bins} rendered at once, we have to upload $2 \times N_{\text{bins}}$ assets.

The first choice, continuous bin rendering, is the best candidate for maximum raw performance, so is the privileged method for offline rendering. The second choice is interesting for interactive rendering when color accuracy is preferred over raw performance to get a fast preview.

42.6 DISCUSSION

In this section we discuss open problems and aspects of spectral rendering and propose some solutions that could be tested and implemented in a production environment.

42.6.1 EFFICIENT SPECTRAL ASSET MANAGEMENT

As we discussed several times in this chapter, GPU memory may be the main constraint when considering spectral rendering. In particular, with scenes containing many textured materials, environment maps, or even tabulated spectral BSDF data, the memory management may be tricky. Uploading back and forth many textures (or buffers) has a significant impact (up to several seconds for big assets). It is always possible to reduce the number of processed wavelengths or to use a subsampling approach, as presented in Section 42.5, to circumvent memory constraint at the cost of rendering efficiency. Another straightforward solution, at least for reflective textures, is to use a better asset representation, namely an analytical one.

At the moment, in Malia, spectral textures are multi-layered images to store the values for all its wavelengths. This brute-force approach ensures that any material with sharp spectral distribution remains accurately depicted. However, many "everyday" materials do not exhibit sharp spectra. In those cases, projecting the spectrum into an appropriate basis (e.g., Jakob and Hanika [12] or Otsu et al. [19] using RGB uplifting, or Peters et al. [21] using bounded reflectance spectra) and reconstructing it on the fly may be sufficient to save memory at the cost of preprocessing the assets. Doing so permits sparing some GPU memory that can be used for assets with sharp distribution (for which uplifting is hardly possible) to their full spectral resolution. Note that for really smooth assets it may also be worth considering plain downsampling when one prefers performance over accuracy.

More extensive studies should be conducted to determine a good trade-off between spectral accuracy (to ensure that all sharp spectral features are depicted) and per-material memory management (full resolution, downsampled representation, RGB uplifting, etc.) with respect to the application considered.

Figure 42-13 illustrates part of this discussion: glasses have an absorption spectrum sharp enough to introduce significant differences when compared to plain RGB rendering, but they can be described analytically by a simple Gaussian, being more memory-friendly. However, the Lucy statue has a dielectric material with a handmade IOR (probably nonexistent but offering artistic features), which has no straightforward lightweight representation due to its sharp spectral distribution. The dragon also exhibits two handmade conductors with sharp spectral distribution. Note that, as illustrated by Figure 42-11, these spectral features are not well depicted in RGB.

42.6.2 DENOISING

Denoising is currently the most popular choice when it comes to real-time Monte Carlo path tracing. Even with an efficient RGB renderer and a high-end desktop computer, it is hard to provide at least 30 converged images per second. However, if a low-sample image (noisy) can be denoised efficiently, this goal can be attained.

The idea also applies to a spectral renderer. One may attempt to denoise a RGB image obtained from a spectral renderer with a real-time-capable RGB denoiser. Though such an approach is beyond the scope of this chapter, we must point out that one must be sure that a RGB algorithm can correctly

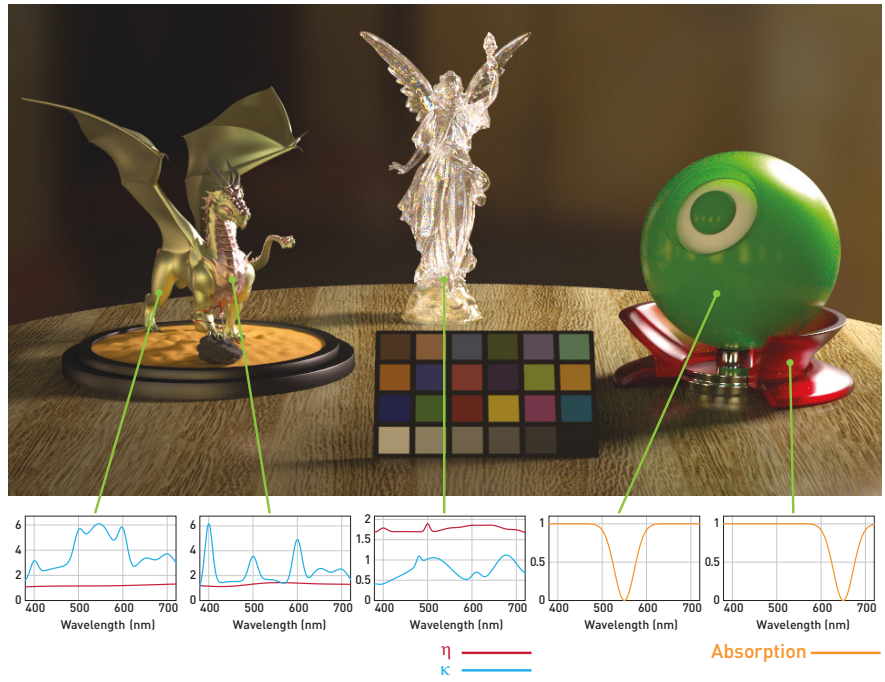


Figure 42-13. *The scene from Figure 42-11, illustrating examples of strong spectral dependency for the dragon, the Lucy statue, and the probe. These sharp distributions may require preprocessing to be stored more efficiently on the GPU, especially if they are set to vary spatially, such as in a texture.*

handle wavelength-dependent features and the potential color noise induced by the spectral rendering process. Further research may be necessary to assess this compatibility, to extend the RGB denoiser to the spectral domain, or to propose a fast and efficient spectral denoiser.

Note that our current concerns with the denoising of spectral images only apply in the context of rendering for predictive purposes. For entertainment purposes, if spectral features are required, one may look at some non-predictive-friendly denoising approaches. Some approaches (e.g., Liu et al. [15]) efficiently denoise a rendering with only one or two rays per pixel but introduce a potential mathematical bias by tweaking light transport integrals for built-in denoising. This kind of approach may be spectral-friendly, as it is incorporated in the light transport itself, but will not be predictive due to the potential bias.

42.7 CONCLUSION AND OUTLOOK

In this chapter, we have introduced spectral rendering and motivated its usage when color accuracy is desired or when specific physical effects need to be handled (e.g., dispersion, polarization, and fluorescence). We have illustrated the limitation of the tristimulus approach for light transport. Furthermore, we have introduced a pipeline for efficient spectral rendering on the GPU that limits memory usage and data exchanges between the device and the host. Finally, we have also introduced some optimizations to improve interactivity while retaining the predictive nature of the renderer.

Due to its still early adoption, spectral rendering is still a niche in the entertainment industry, but has become increasingly popular in recent years thanks to increased computational power and memory. We believe that spectral rendering will also generate more interest for GPU rendering despite the challenges it brings in terms of memory and bandwidth. GPU rendering offers an efficient way for providing both spectral and interactivity features in the context of predictive rendering. Recent academic-oriented spectral renderers have already opened this lead.

ACKNOWLEDGMENTS

This project was supported by the Agence Nationale de la Recherche Project VIDA (ANR-17-CE23-0017) as well as the Regional Nouvelle-Aquitaine Grant SIMOREVA-360. High dynamic range (HDR) environment maps were freely provided by HDRI Haven [27] and converted to spectral HDR images with our own tool. The original *bathroom* scene was modeled by McGuire [17]. Its materials were adapted and modified to transform the RGB scene into a spectral one. The dragon was freely downloaded from Bitterli's resources [2].

REFERENCES

[1] Antwerpen, D. v., Seibert, D., and Keller, A. A simple load-balancing scheme with high scaling efficiency. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 127–133. Apress, 2019. DOI: 10.1007/978-1-4842-4427-2_10.

[2] Bitterli, B. Rendering resources. https://benedikt-bitterli.me/resources/, 2016.

[3] Blender Foundation. Cycles. https://www.cycles-renderer.org/, 2021.

[4] Caredda, C., Mahieu-Williame, L., Sablong, R., Sdika, M., Guyotat, J., and Montcel, B. Optimal spectral combination of a hyperspectral camera for intraoperative hemodynamic and metabolic brain mapping. *Applied Sciences*, 10(15):5158:1–5158:23, 2020. DOI: 10.3390/app10155158.

[5] Dufay, A., Murray, D., Pacanowski, R., et al. The Malia rendering framework. https://pacanows.gitlabpages.inria.fr/MRF, 2019.

[6] Evans, G. F. and McCool, M. D. Stratified wavelength clusters for efficient spectral Monte Carlo rendering. In *Proceedings of the Graphics Interface 1999 Conference, June 2-4, 1999, Kingston, Ontario, Canada*, pages 42–49, 1999. http://graphicsinterface.org/wp-content/uploads/gi1999-7.pdf.

[7] Fascione, L., Hanika, J., Fajardo, M., Christensen, P., Burley, B., and Green, B. Path tracing in production—Part 1: Production renderers. In *ACM SIGGRAPH 2017 Courses*, 13:1–13:39, 2017. DOI: 10.1145/3084873.3084904.

[8] Fascione, L., Hanika, J., Leone, M., Droske, M., Schwarzhaupt, J., Davidovic, T., Weidlich, A., and Meng, J. Manuka: A batch-shading architecture for spectral path tracing in movie production. *ACM Transactions on Graphics*, 37(3):31:1–31:18, 2018. DOI: 10.1145/3182161.

[9] Guo, J. J., Bauszat, P., Bikker, J., and Eisemann, E. Primary sample space path guiding. In *Proceedings of the Eurographics Symposium on Rendering: Experimental Ideas and Implementations*, pages 73–82, 2018. DOI: 10.2312/sre.20181174.

[10] Hecht, E. *Optics*. Pearson, 5th edition, 2016.

[11] Jakob, W. Mitsuba 2: Physically based renderer. http://www.mitsuba-renderer.org, 2010.

[12] Jakob, W. and Hanika, J. A low-dimensional function space for efficient spectral upsampling. *Computer Graphics Forum (Proceedings of Eurographics)*, 38(2):147–155, Mar. 2019. DOI: 10.1111/cgf.13626.

[13] Joannopoulos, J. D., Johnson, S. G., Winn, J. N., and Meade, R. D. *Photonic Crystals: Molding the Flow of Light*. Princeton University Press, 2nd edition, 2008.

[14] Keller, A., Wächter, C., Raab, M., Seibert, D., van Antwerpen, D., Korndörfer, J., and Kettner, L. The iray light transport simulation and rendering system. In *ACM SIGGRAPH 2017 Talks*, 34:1–34:2, 2017. DOI: 10.1145/3084363.3085050.

[15] Liu, E., Llamas, I., Cañada, J., and Kelly, P. Cinematic rendering in UE4 with real-time ray tracing and denoising. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 289–319. Apress, 2019. DOI: 10.1007/978-1-4842-4427-2_19.

[16] Mayerhöfer, T. G., Pahlow, S., and Popp, J. The Bouguer–Beer–Lambert law: Shining light on the obscure. *ChemPhysChem*, 21(18):2029–2046, 2020. DOI: https://doi.org/10.1002/cphc.202000464.

[17] McGuire, M. Computer graphics archive. https://casual-effects.com/data, 2017.

[18] Nimier-David, M., Vicini, D., Zeltner, T., and Jakob, W. Mitsuba 2: A retargetable forward and inverse renderer. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 38(6):203:1–203:17, Dec. 2019. DOI: 10.1145/3355089.3356498.

[19] Otsu, H., Yamamoto, M., and Hachisuka, T. Reproducing spectral reflectances from tristimulus colours. *Computer Graphics Forum*, 37(6):370–381, 2018. DOI: https://doi.org/10.1111/cgf.13332.

[20] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics*, 29(4):66:1–66:13, July 2010. DOI: 10.1145/1778765.1778803.

[21] Peters, C., Merzbach, S., Hanika, J., and Dachsbacher, C. Using moments to represent bounded signals for spectral rendering. *ACM Transactions on Graphics*, 38(4):136:1–136:14, July 2019. DOI: 10.1145/3306346.3322964.

[22] Pharr, M. PBRT version 4. https://github.com/mmp/pbrt-v4, 2020.

[23] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 3rd edition, 2016.

[24] Subtil, N. and Werness, E. NVIDIA RTX: Enabling Ray Tracing in Vulkan. Presentation at GPU Technology Conference, https://on-demand.gputechconf.com/gtc/2018/presentation/s8521-advanced-graphics-extensions-for-vulkan.pdf, March 27, 2018.

[25] The ART development team. The Advanced Rendering Toolkit. https://cgg.mff.cuni.cz/ART, 2018.

[26] Wilkie, A., Nawaz, S., Droske, M., Weidlich, A., and Hanika, J. Hero wavelength spectral sampling. In *Proceedings of the 25th Eurographics Symposium on Rendering*, pages 123–131, 2014. DOI: 10.1111/cgf.12419.

[27] Zaal, G., Majboroda, S., and Mischok, A. HDRI Haven. https://hdrihaven.com/.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 43

EFFICIENT UNBIASED VOLUME PATH TRACING ON THE GPU

Nikolai Hofmann and Alex Evans
NVIDIA

ABSTRACT

We present a set of optimizations that improve the performance of high-quality volumetric path tracing. We build upon unbiased volume sampling techniques, i.e., null-collision trackers [16, 8, 15], with voxel data stored in an OpenVDB [14, 13] tree. The presented optimizations achieve an overall 2× to 3× speedup when implemented on a modern GPU, with an approximately 6.5× reduction in memory footprint. The improvements primarily stem from a multi-level digital differential analyzer (DDA) [1, 11, 6] to step through a grid of precomputed bounds; a replacement of the top levels of the OpenVDB tree with a dense indirection texture, similar to virtual textures [3, 4, 17], while preserving some sparsity; and quantization of the voxel data, encoded using GPU-supported block compression. Finally, we examine the isolated effect of our optimizations, covering stochastic filtering, the use of dense indirection textures, compressed voxel data, and single- versus multi-level DDAs.



Figure 43-1. *The Disney Moana cloud [18] (left) and an explosion cloud with emission [7] (right) with three bounces of multiple scattering, rendered in 3.9 ms and 4.1 ms per sample, respectively, using our approach at* 1920 × 1080 *resolution on an NVIDIA RTX 3090. Our optimizations and lossy compression scheme provide a 2.5× to 3.0× speedup and a ca. 6.5× reduced memory footprint over the baseline implementation, without noticeable loss in quality (PSNR > 50 dB).*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_43

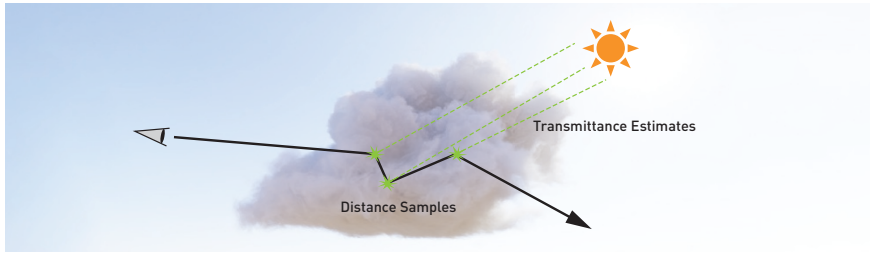


Figure 43-2. *An example path is traced through a volume: We use distance sampling to find the collision vertices along the main path. At each vertex, transmittance estimation computes the fraction of photons arriving from the light source, sampled along shadow rays.*

43.1 BACKGROUND

Rendering volumetric data, e.g., as in Figure 43-1, is a memory- and compute-intensive task. We refer the reader to the state of the art report by Novák et al. [15] for an extensive overview. The cloud in the left image is represented by a $1000 \times 680 \times 1224$ grid of density values. Storing a 32-bit floating-point value for every voxel in the grid would require 3.3 GB of storage. OpenVDB [14] uses a tree structure to compactly encode constant regions of the volume, requiring only 585.2 MB to encode the cloud. By quantizing the data carefully, we build on the serialized OpenVDB representation (NanoVDB [12]) and re-encode it into three textures whose combined size is 86.7 MB, at the cost of some quantization error. The error is small enough to not be visible in the rendered results, which will be discussed in Section 43.5.3, while the performance of random volume lookups is improved.

The two fundamental queries that our data structure must efficiently support are *distance sampling* and *transmittance estimation*. The former is used to determine the length of each path segment between scattering events in the volume, whereas the latter computes the probability of a ray segment passing through the volume without interacting with it. The *transmittance* value is used to estimate the fractional visibility along shadow rays. See Figure 43-2 for an illustration applied to path tracing.

In homogeneous media with constant density μ_c, an unbiased distance sampler is straightforward: we importance-sample the *free-flight* distance t according to transmittance with a cumulative distribution function (the Beer–Lambert law [9]):

$$F(t) = 1 - e^{-\mu_c t}. \tag{43.1}$$

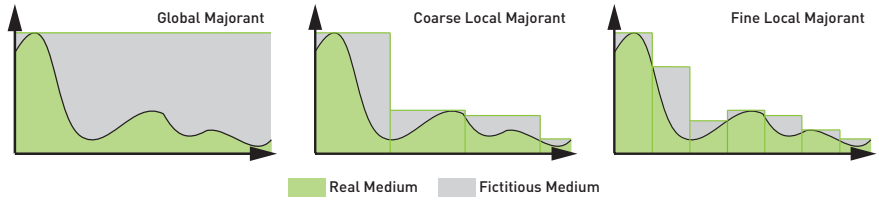


Figure 43-3. *Local versus global upper bounds: Null-collision methods homogenize regions of the volume by introducing a fictitious medium to bulk out the density to be piecewise constant in each region. The more finely we subdivide, the tighter these majorants become, and the less fictitious matter is added.*

Thus, we can sample free paths with uniform random numbers $\xi \in [0, 1)$, such that

$$t = -\frac{\ln(1 - \xi)}{\mu_c}. \tag{43.2}$$

For spatially varying (heterogeneous) volumes, we must extend this sampling scheme because the previous approach is only valid for constant density. We rely on delta tracking [19] for free path sampling and ratio tracking [16] for transmittance estimation. The main idea behind both approaches is to homogenize the medium by introducing additional fictitious matter, which has no impact on light transport. In other words, we "fill" all gaps in the medium between the maximum (*majorant*) $\mu_{\max}$ and real density μ_r with *null-collision density*: $\mu_n(x) = \mu_{\max} - \mu_r(x)$. This allows us to analytically sample the free-flight distances as before, as if the volume were homogeneous.

In order to return to the correct distribution, each collision must now be stochastically categorized as *null* or *real*, in proportion to the ratio of null to real matter:

$$p_{\text{null}} = \frac{\mu_n(x)}{\mu_r(x) + \mu_n(x)}. \tag{43.3}$$

We wish to minimize p_{null}, as such collisions are "wasted effort" and have no effect on light transport. This corresponds to minimizing $\mu_{\max}$: instead of using a single global maximum, we subdivide space into smaller regions and precompute a local $\mu_{\max}$ for each region, which more closely follows $\mu_r(x)$. See Figure 43-3 for an illustration. Though this leads to fewer rejected null collisions, the ray must be split at each region boundary, so as not to leave the region for which each upper bound was computed. This splitting introduces runtime overhead that offsets the benefit of the tighter bounds.

To render a single path-traced frame at typical resolutions and sample counts, these algorithms require hundreds of millions of volume density

evaluations. To make these as efficient as possible, in both execution time and memory bandwidth use, we next present a GPU-friendly data structure for efficiently storing and querying the volume data.

43.2 COMPRESSED DATA STRUCTURE

We use three textures to sparsely store the volume data: an *atlas texture*, *indirection texture*, and min/max *range texture*. Similar to Crassin et. al. [4], and following the leaf size in NanoVDB [12], we refer to a group of $8 \times 8 \times 8$ voxels containing data, e.g., density values, as a *brick*. We pack the bricks into the atlas texture. Offsets into the atlas are stored in a dense indirection texture, which serves to link a position in space with a brick of data in the atlas, similar to a page table. This scheme is applied to address sparsity and is also known as *virtual textures* [17, 10]. We precompute and store in the range texture the minimum and maximum value of each brick's voxels (referred to as *minorant* and *majorant*, respectively), using these ranges to scale the values in the atlas texture to lie in [0, 1]. This makes it straightforward to encode high dynamic range values in the atlas at reduced bit depths, such as 16 or 8 bits in `UNORM` format, or 4 bits via hardware-supported block compression (`BC4`). In regions of constant density (minorant = majorant), including empty space, the brick's range will be zero and we can therefore omit storing any unique data in the atlas for that brick. The indirection texture simply points such bricks at address 0, whose contents will be ignored due to the empty range. A visual overview of our data structure is given in Figure 43-4. Note that all code is presented in voxel index space, where one unit along any axis corresponds to the distance between adjacent voxels. A volume lookup thus translates to the lines of HLSL code in Listing 43-1.

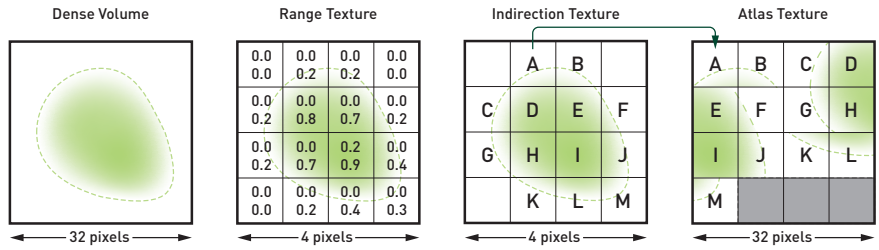


Figure 43-4. *We illustrate our data structure with a 2D example. The dense volume (left) is divided into 8^3 (8^2 in the figure) bricks packed into an atlas texture (right). A range texture encodes the range of values in each brick. In this example, three bricks have empty ranges and are thus omitted from the atlas. An indirection texture encodes the position of each brick in the atlas.*

Listing 43-1. *Sparse volume lookup routine (HLSL).*

```
float lookup(int3 index) {
    int3 brick = index >> 3;
    float2 range = rangetex[brick]; // (minorant, majorant)
    uint3 ptr = indirecttex[brick].xyz;
    float value_unorm = atlastex[(ptr.xyz << 3) + (index & 7)].x;
    return value_unorm * (range.y - range.x) + range.x;
}
```

We encode the indirection texture as `RGBA8Uint`, the range texture as `RGFloat16`, and the atlas texture via `BC4` hardware compression. Therefore, the total memory cost of our encoding scheme is $\frac{W \times H \times D}{64} + (N \times 256)$ bytes, where N is the number of spatially varying bricks within the volume. Because we lossily quantize the voxel data, our format is less suitable for storage or simulation. However, for rendering of density fields, the brick-level rescaling ensures that the visual impact is minimal, which will be evaluated later in Section 43.5.3. Although the leaf data is $8\times$ more compact than the equivalent data stored as `float32` values in a NanoVDB tree, the latter's tree structure is more compact than our dense indirection and range texture scheme. Thus, the overall memory saving amounts to ca. $6.5\times$. If the volume is very sparse, a second level of indirection, further mirroring the NanoVDB tree structure, could reduce the storage requirements further, at the cost of complicating the lookup routine.

43.3 FILTERING AND RANGE DILATION

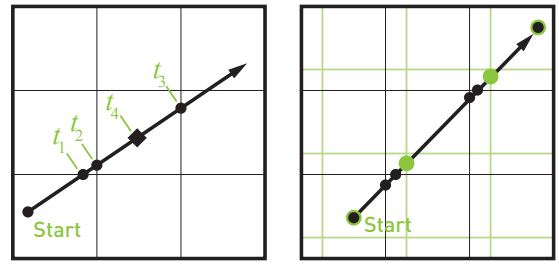
A common approach to smoothly interpolate the discrete density grid is trilinear or tricubic interpolation. Performing this operation in software has significant cost, but the GPU texture unit only supports trilinear filtering for directly addressable, i.e., dense, textures. We could work around that limitation by redundantly storing voxels that lie on the border of each brick [4]. However, that would impose either a $\frac{8^3}{7^3} \approx 1.49\times$ memory overhead due to duplicated borders or a performance penalty from falling back to software filtering at those locations. We instead stochastically jitter the lookup point by ± 0.5 voxels before point sampling. This is equivalent to importance sampling linear interpolation, where the probability density function and the interpolation factor cancel out. At high sample counts, this "stochastic filtering" scheme yields identical results to trilinear filtering at a fraction of the cost. Refer to Section 43.5.2 for an evaluation of the noise introduced.

Whatever the chosen filtering scheme, values from neighboring bricks may "leak" into each other due to the width of the filter kernel, causing filtered values to lie outside the central brick's precomputed range. Ratio tracking requires the per-brick majorants to be strict upper bounds [16]. Even applying the computationally more expensive weighted ratio tracker [8], which relaxes that requirement, may exhibit increased variance if the majorant is not conservative. To address this, we precompute the brick majorants over a 10^3 volume, encompassing the one-ring of voxels around the brick's original 8^3 footprint. For a trilinear or stochastic trilinear filter, which only have a kernel radius of half a voxel, this allows the center of the jittered sample to lie an extra half voxel outside the brick. We exploit this slight "over-dilation" to take larger steps in the digital differential analyzer (DDA) than would otherwise be possible, which we will describe next.

43.4 DDA TRAVERSAL

Both the distance sampling and transmittance estimation schemes require us to trace rays through the volume to find collisions. We start by sampling a target optical thickness: $\tau_{\text{target}} = -\ln(1 - \xi)$. We then employ a DDA [1] to visit all bricks that intersect the ray and accumulate their (tentative) optical thicknesses $\tau_{\text{brick}} = \mu_{\text{brick_max}} \times dt$, where dt is the length of the ray segment that intersects the brick. Should the accumulated optical thickness exceed the sampled target thickness, we step backward along the ray until they match exactly. This is illustrated in Figure 43-5a. At this point, we have found a tentative collision. Next, we perform a filtered density lookup and compare it to the local majorant, stochastically deciding whether to accept the collision as real. When sampling distances, we return the distance to the first real collision along the ray. When estimating transmittance, we adjust the current estimate using the ratio of real to fictitious matter. In the event of a rejected null collision, we restart the traversal by sampling a new target optical thickness value. The algorithm continues until we have sampled a distance, the transmittance estimate is aborted by Russian roulette, or we have left the volume.

The key to achieving optimal traversal performance is the balance between large step sizes and tight upper bounds. A small brick size yields accurate upper bounds and thus less wasted samples, but causes more work, splitting the ray due to the DDA's small steps. On the other hand, large bricks enable the DDA to move quickly, but cause a higher sample rejection rate. See Figure 43-3. A rejected sample is costly because it corresponds to multiple



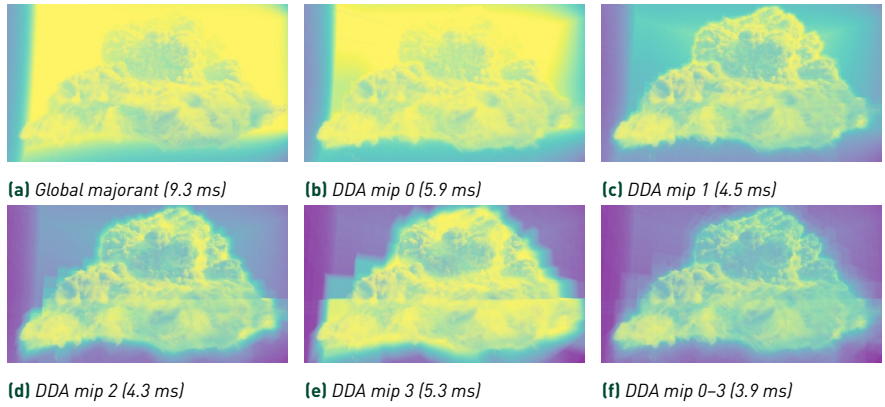
(a) *Regular DDA* **(b)** *Brick-dilated DDA*

Figure 43-5. *Left: the DDA traverses along the ray in steps (t_1, t_2, t_3) such that each segment falls entirely inside a brick. The majorants τ_{brick} of each brick are accumulated, $\tau_{\text{sum}} = \tau_1 + \tau_2 + \tau_3$, until their sum exceeds τ_{target}. Finally, a backward step (t_4) is taken to the point where τ_{sum} matches τ_{target} exactly. Right: during traversal, a regular DDA would require five steps, as highlighted by the black intersection points. Because we intersect against dilated brick boundaries instead, 0.5 voxels outside in the direction of travel, our DDA manages in only three steps, as shown by the green intersection points. This also improves numerical stability, as we do not land exactly on brick boundaries.*

texture lookups and requires us to restart the traversal after reverting to the point of collision, which voids some of the benefits of larger DDA steps.

Therefore, we additionally compute three min-max mip levels of the range texture. This enables us to search for tentative collisions at four different scales, with minorant and majorant information available for dilated bricks computed over a domain of $8 \times 2^{\text{mip}} + 2$ voxels, with mip $\in [0, 3]$. As we step backward to the exact point of collision in a brick, the chosen mip level only affects how quickly collisions are found, without introducing bias or affecting image quality.

The chosen mip level has a large impact on performance and varies strongly between scenes. It would be possible to implement each level of the DDA as a separate loop in a shader, for example, starting with a coarse mip level loop and then falling down to finer levels in separate loops. However, on a wide *single-instruction-multiple-threads* (SIMT) device like a GPU, this would lead to significant execution divergence between rays at different levels of the hierarchy. We therefore chose to reformulate the DDA to be able to independently select a mip level at every step without branching, a concept previously applied to screen-space reflections [6]. We empirically chose a very straightforward heuristic for selecting the per-step mip level: We initialize to the coarsest level (mip = 3.0), add 0.33 for each iteration, and subtract 2.0 when a tentative collision was found. The DDA then takes a step according to



(a) *Global majorant (9.3 ms)* **(b)** *DDA mip 0 (5.9 ms)* **(c)** *DDA mip 1 (4.5 ms)*

(d) *DDA mip 2 (4.3 ms)* **(e)** *DDA mip 3 (5.3 ms)* **(f)** *DDA mip 0–3 (3.9 ms)*

Figure 43-6. *Average number of volume and majorant lookups per path on the same viewpoint as Figure 43-1, mapped from* [0, 150]. *The number of required volume lookups can be reduced significantly, especially for empty space regions, when stepping over a larger grid of upper bounds [(a) versus (d)]. However, when choosing the mip level too large, and thus causing frequent backtracking, the number of lookups increases (e). Our multi-level DDA addresses this issue by adapting the step size locally and manages with the least amount of lookups overall (f). Note that in (a) all lookups are full volume lookups, while for the DDA-based variants most are solely majorant lookups, which only require one instead of three texture taps.*

the nearest integer mip level: $\lfloor \text{mip} \rceil$. Figure 43-6 illustrates how the number of volume lookups is affected by the DDA mip level. The multi-level traversal scheme achieves the best performance, making the least number of lookups overall, and automatically adapts to the underlying scene.

As previously mentioned in Section 43.3, filtering requires us to use dilated majorants, which we "over-dilate" by half a voxel. This has the additional benefit of allowing us to perform ray intersections against bricks that have been enlarged by half a voxel, reducing the number of steps taken while still leaving room for the filter's footprint. Because the DDA oversteps into each brick by half a voxel, we eliminate the usual numerical problems associated with intersections falling exactly on brick boundaries. We illustrate this in Figure 43-5b and provide HLSL code snippets for the core parts of our algorithm in Listings 43-2 and 43-3.

Listing 43-2. *DDA stepping routine using bricks enlarged by half a voxel (HLSL).*

```
float stepDDA(float3 ro, float3 ri, float3 pos, int mip) {
    const float dim = 8 << mip;
    const float3 ofs = ri * (((ri >= 0.f) ? dim + 0.5f : -0.5f ) - ro);
    const float3 tmax = floor(pos * (1.f / dim)) * dim * ri + ofs;
    return min(tmax.x, min(tmax.y, tmax.z));
}
```

Listing 43-3. *DDA-based volume sampling routine (HLSL). The two red lines are all that is required to dynamically select a mip level at each step.*

```
float tauToGo = -log(1.f - random());
float mip = 3;
float3 invRayDir = 1.0 / rayDir;
while (/* Inside volume */) {
    float3 pos = rayOrigin + t * rayDir;
    float majorant = lookupMajorant(pos, round(mip));
    float nextt = stepDDA(rayOrigin, invRayDir, pos, round(mip));
    mip = min(3.f, mip + 0.33);
    float dt = nextt - t;
    float dtau = majorant * dt;
    t = nextt;
    tauToGo -= dtau;
    if (tauToGo > 0) continue;          // No collision, skip ahead
    mip = max(0.f, mip - 2);
    t += dt * tauToGo / dtau;           // Step back to collision
    float density = lookup(rayOrigin + t * rayDir);
    if (random() * majorant < density) { /* Handle real collision */ }
    tauToGo = -log(1.f - random());     // Null collision, continue
}
```

43.5 RESULTS

In the following, we evaluate our approach against a baseline implementation and examine the achieved speedups and compression rates versus the magnitude of numerical error introduced into the image. We summarize our results in Table 43-1.

43.5.1 BASELINE

As a baseline, we employ an unbiased path tracer using NanoVDB to lookup `float32` density values, delta tracking for distance sampling, and ratio tracking for transmittance estimation, all with global majorants. We trace up to three bounces of multiple scattering, while shooting a shadow ray at each path vertex, importance sampling the scene's light sources, and weighting accidental and sampled light source hits from next event estimation with multiple importance sampling. We implement anisotropic scattering via the Henyey–Greenstein [5] phase function, although we used $g = 0$ in our evaluation, i.e., isotropic scattering. As a quality baseline, we rely on a software implementation of trilinear filtering to smooth the discrete, regular grids. For the performance baseline, we chose stochastic filtering over trilinear, due to its substantial effect on performance. Any errors introduced into the image are evaluated using the ꟻLIP [2] perceptual metric (see also Chapter 19) and the peak signal-to-noise ratio (PSNR), by comparing converged images with 32,000 samples per pixel.

Moana Cloud (Figure 43-1, left)						
Variant	**Time**	**MB**	**÷ Time**	**× MB**	**ꟻLIP**	**PSNR**
BaselineQ (trilinear)	38.9	585.2	0.29	1.00	0.00000	+inf
BaselineP (stochastic)	11.1	585.2	1.00	1.00	0.00501	56.23 dB
Texture (16 bit)	9.2	292.3	1.21	0.50	0.00501	56.23 dB
Texture (8 bit)	9.1	153.9	1.22	0.26	0.00589	55.89 dB
Texture (BC4)	9.2	86.7	1.21	0.15	0.00738	54.34 dB
BC4, DDA mip 0	5.9	86.7	1.88	0.15	0.00811	51.99 dB
BC4, DDA mip 1	4.5	86.7	2.47	0.15	0.00806	52.08 dB
BC4, DDA mip 2	4.3	86.7	2.58	0.15	0.00805	52.12 dB
BC4, DDA mip 3	5.3	86.7	2.09	0.15	0.00804	52.16 dB
BC4, DDA adaptive	3.9	86.7	2.85	0.15	0.00809	52.07 dB

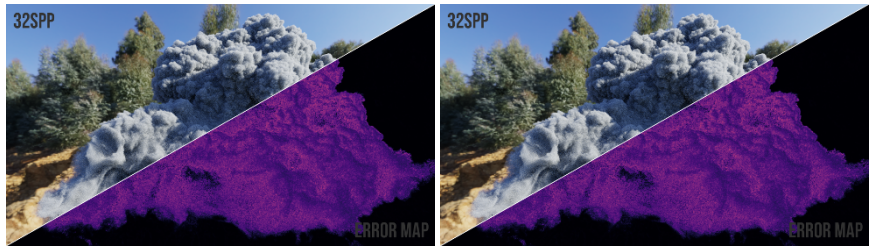
Table 43-1. *Timings in milliseconds are taken on an NVIDIA RTX 3090 GPU and refer to rendering one sample per pixel at* 1920 × 1080 *resolution, with three bounces of multiple (isotropic) scattering. We use the Moana cloud [18] at* 1000 × 680 × 1224 *voxel resolution. As a baseline, we employ NanoVDB in* `float32` *encoding with trilinear filtering (BaselineQ) for quality and stochastic filtering (BaselineP) for performance. We compare performance and memory consumption between the baseline and several variations of our approach with different levels of quantization and DDA traversal strategies. ꟻLIP [2] and PSNR are computed from converged images with 32,000 samples.*

43.5.2 STOCHASTIC SAMPLING

Switching from eight weighted volume lookups (trilinear interpolation) to a single stochastically jittered sample gives the single largest speedup of our optimizations, roughly 3.5×. It introduces no bias in the converged image and thus provides a virtually indistinguishable result (PSNR 56.23 dB). Even at low sample counts, the difference is negligible. When comparing both software-trilinear and stochastic-trilinear filtering at 32 samples per pixel to a converged result, we only found a less than 0.1 dB PSNR difference between the two approaches, as outlined in Figure 43-7.

43.5.3 QUANTIZED TEXTURE REPRESENTATION

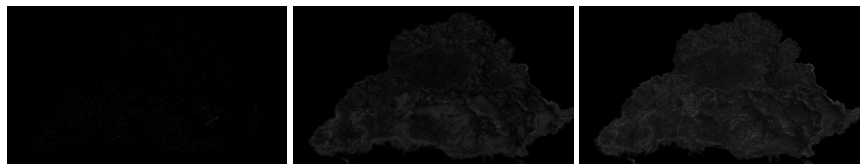
We re-encode the NanoVDB `float32` data into textures as previously described in Section 43.2. We examine the error due to quantization for 16 bits per texel (`UNORM16`), 8 bits per texel (`UNORM8`), and 4 bits per texel (`BC4`) in an isolated environment in Figure 43-8. The most compressed form (`BC4`) is selected for all subsequent steps, as it yields the most compact representation (6.5× compression) without notable quality or performance impacts. The high quality regardless of texture compression is a valuable



(a) *FLIP error map, software-trilinear, FLIP: 0.13909, PSNR: 23.05 dB*

(b) *FLIP error map, stochastic-trilinear, FLIP: 0.13937, PSNR: 22.97 dB*

Figure 43-7. *Virtually indistinguishable results from (b) our stochastic filtering scheme, compared to (a) the trilinear baseline implementation at 32 samples per pixel. We compute FLIP [2] and PSNR against a converged reference rendering with 32,000 samples. Note that the version with stochastic filtering rendered roughly 3.5× as fast.*



(a) *Error × 10 for 16 bit, FLIP: 0.00048, PSNR: 75.10dB*

(b) *Error × 10 for 8 bit, FLIP: 0.00437, PSNR: 60.72dB*

(c) *Error × 10 for BC4, FLIP: 0.00682, PSNR: 56.15dB*

Figure 43-8. *Isolated numerical error introduced by our lossy compression, evaluated against uncompressed 32-bit encoding, using global majorants and stochastic filtering. We evaluated FLIP [2] and PSNR on converged images with 32,000 samples. For visibility, we show a grayscale FLIP error map of the cloud in Figure 43-1 scaled by 10×, because the error would otherwise be imperceptible.*

result, as memory is often a limiting factor in volume rendering, particularly when rendering volumetric animations with many frames resident in memory.

43.5.4 SINGLE- AND MULTI-LEVEL DDA WITH LOCAL MAJORANTS

As demonstrated in Figure 43-6, the resolution of the local majorants are inherently tied to rendering performance. A coarser grid enables taking larger steps, but also reduces the tightness of the bounds and may lead to backtracking. We measure timings for different mip levels of the range texture, yielding local majorants computed over $8 \times 2^{\text{level}} + 2$ texel bricks for level $\in [0, 3]$. We found the optimal level to be highly scene dependent, which turned out to be level = 2 for the Moana cloud data set (Figure 43-6d). By dynamically choosing a mip level per step, as we described in Section 43.4, we manage to achieve the largest speedup and automatically accommodate different scene sizes as well.

43.6 CONCLUSION

In summary, we showed how to optimize existing unbiased volumetric path tracing techniques for use on modern GPUs and managed to achieve a 2× to 3× speedup at a roughly 6.5× reduced memory footprint compared to a baseline implementation. We demonstrated that the error introduced by our lossy compression scheme is virtually imperceptible and thus enables high-quality renderings of volumetric assets at a fraction of the cost.

REFERENCES

[1] Amanatides, J. and Woo, A. A fast voxel traversal algorithm for ray tracing. In *Eurographics 1987—Technical Papers*, pages 3–10, 1987. DOI: 10.2312/egtp.19871000.

[2] Andersson, P., Nilsson, J., Akenine-Möller, T., Oskarsson, M., Åström, K., and Fairchild, M. D. ꟻLIP: A difference evaluator for alternating images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques (HPG 2020)*, 3(2):15:1–15:23, 2020. DOI: 10.1145/3406183.

[3] Beyer, J., Hadwiger, M., Möller, T., and Fritz, L. Smooth mixed-resolution GPU volume rendering. In *Proceedings of the Fifth Eurographics/IEEE VGTC Conference on Point-Based Graphics*, pages 163–170, 2008.

[4] Crassin, C., Neyret, F., Lefebvre, S., and Eisemann, E. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, pages 15–22, 2009.

[5] Henyey, L. G. and Greenstein, J. L. Diffuse radiation in the galaxy. *The Astrophysical Journal*, 93:70–83, 1941.

[6] Hofmann, N., Bogendörfer, P., Stamminger, M., and Selgrad, K. Hierarchical multi-layer screen-space ray tracing. In *Proceedings of High Performance Graphics*, pages 1–10. 2017.

[7] JangaFX. Free VDB simulations created with EmberGen. https://jangafx.com/software/embergen/download/free-vdb-animations/, 2020. Acessed December 10, 2020.

[8] Kutz, P., Habel, R., Li, Y. K., and Novák, J. Spectral and decomposition tracking for rendering heterogeneous volumes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2017)*, 36(4):111:1–111:16, 2017. DOI: 10.1145/3072959.3073665.

[9] Lambert, J. H. *Photometria sive de mensura et gradibus luminis, colorum et umbrae*. Klett, 1760.

[10] Lefohn, A. E., Sengupta, S., Kniss, J., Strzodka, R., and Owens, J. D. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, 2006. DOI: 10.1145/1122501.1122505.

[11] Museth, K. Hierarchical digital differential analyzer for efficient ray-marching in openvdb. In *ACM SIGGRAPH 2014 Talks*, 40:1. 2014. DOI: 10.1145/2614106.2614136.

[12] Museth, K. NanoVDB: A GPU-friendly and portable VDB data structure for real-time rendering and simulation. In *ACM SIGGRAPH 2021 Talks*, 2021. In submission.

[13] Museth, K. VDB: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics*, 32(3):27:1–27:22, 2013. DOI: 10.1145/2487228.2487235.

[14] Museth, K., Lait, J., Johanson, J., Budsberg, J., Henderson, R., Alden, M., Cucka, P., Hill, D., and Pearce, A. OpenVDB: An open-source data structure and toolkit for high-resolution volumes. In *ACM SIGGRAPH 2013 Courses*, 19:1. 2013. DOI: 10.1145/2504435.2504454.

[15] Novák, J., Georgiev, I., Hanika, J., and Jarosz, W. Monte Carlo methods for volumetric light transport simulation. *Computer Graphics Forum (Proceedings of Eurographics—State of the Art Reports)*, 37(2):551–576, May 2018. DOI: 10.1111/cgf.13383.

[16] Novák, J., Selle, A., and Jarosz, W. Residual ratio tracking for estimating attenuation in participating media. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2014)*, 33(6):179:1–179:11, Nov. 2014. DOI: 10.1145/2661229.2661292.

[17] Van Waveren, J. M. P. and Hart, E. Using virtual texturing to handle massive texture data. Presentation at GPU Technology Conference, https://www.nvidia.com/content/GTC-2010/pdfs/2152_GTC2010.pdf, September 21, 2010.

[18] Walt Disney Animation Studios. Data sets: Clouds. https://www.disneyanimation.com/data-sets/?drawer=/resources/clouds/, 2020. Acessed December 10, 2020.

[19] Woodcock, E, Murphy, T, Hemmings, P, and Longworth, S. Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry. In *Applications of Computing Methods to Reactor Problems, ANL-7050*, page 557, 1965.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 44

PATH TRACING RBF PARTICLE VOLUMES

Aaron Knoll, Gregory P. Johnson, and Johannes Meng
Intel Corporation

ABSTRACT

Particle volume data are common in scientific computing applications such as cosmology and molecular dynamics, and increasingly in smoke and fluid animations. Path tracing particle data directly has historically proven challenging; most existing approaches rely on conversion to structured volumes, proxy geometry such as slicing or splatting, or special techniques optimized for ray casting but not path tracing. However, it remains desirable to render particles without simplification and with full control over the volume reconstruction filter, in a manner consistent with increasingly ubiquitous ray tracing APIs. In this chapter, we detail a method for quickly traversing, sampling, and path tracing particle volumes using a radial basis function (RBF) model, implemented using the Intel® Open Volume Kernel Library (VKL) and OSPRay rendering framework but suited for general ray tracing hardware and software APIs.

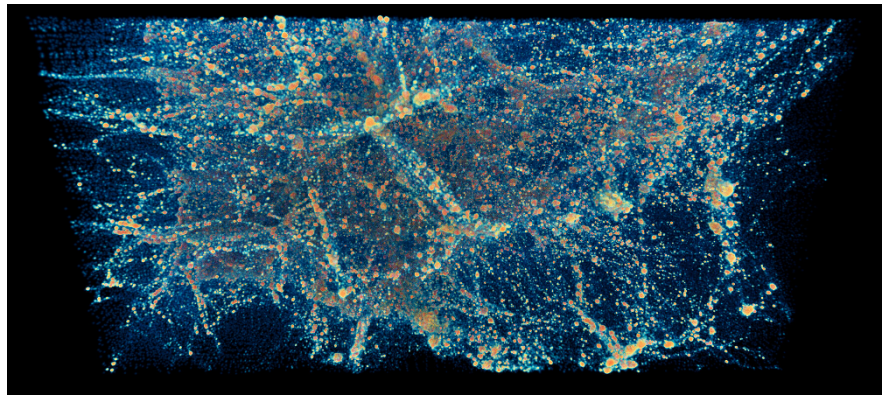


Figure 44-1. *Five million particle HACC [4] cosmology particle data set, path traced with Intel Open VKL at 1.8 FPS on an Intel Xeon 8180M, 28 cores, 2.5 GHz.*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_44

44.1 INTRODUCTION

Particle rendering has diverse origins, applications, and needs. For effects in games, the most popular approach remains to resample onto axis-aligned texture slices (e.g., [3]). In film, it is typical to compute and render the boundary mesh from large particle fluid simulations using marching cubes [12] or more sophisticated techniques. In scientific visualization, different techniques may be appropriate depending on the goal. For example, computational fluid dynamics data are most frequently converted to structured volumes, then isosurfaced or direct volume rendered. Large molecular data are frequently represented as explicit spherical glyphs or isosurfaces and either ray traced or rasterized. Volume splatting [18] projects particles into 2D billboards with an elliptical footprint. It scales as well as the underlying rendering technique—traditionally rasterization.

In many cases, it is desirable to render particle data as actual particles, or to model them as a volume composed of overlapping radial basis functions (RBFs). Modern gravitational N-body cosmology simulations employ millions to trillions of particles, and large molecular dynamics simulations consist of millions to billions of atoms. It is helpful to visually resolve these simulations down to their individual computational elements, without simplification. In the previous *Ray Tracing Gems* book, Knoll et al. [10] showed how to leverage a ray tracing framework to more effectively render large, sparse particle volume data using splatting. This allowed for hundreds of millions of individual particles to be splatted interactively at 4000 resolution, with no simplification, proxy geometry, or filtering of the original data. This approach could be limiting; closely spaced "overlapping" particles resulted in both visual clutter and pathological performance with bounding volume hierarchy (BVH)–based ray tracing methods. Grouping and sorting a subset of particles by depth dramatically improved performance but limited the rendering to primary ray casting as opposed to path tracing.

Volume rendering with a radial basis function model solves the overlap issue by treating the whole volume as one scalar field composed of additive weighted kernels. This method is frequently referred to as a *smoothed particle hydrodynamics* (SPH) approach due to its use in visualizing SPH fluid dynamics codes. With RBF volume rendering, in contrast to splatting, one can adjust the radius and weight of the kernels to achieve a smoothing or sharpening filter over particles as desired. Previous implementations (e.g., [14, 11]) employed optimizations that restricted them to ray casting. In this chapter, we

show how to efficiently path-trace RBF volumes. Our renderer is similar to the Woodcock-tracking volume path tracer of Raab [16], except we implement RBF reconstruction within a volume sampling kernel API (Intel® Open Volume Kernel Library [7]), leverage interval iterators for space skipping, and allow for abstraction of the rendering method (ray casting, ray casting with ambient occlusion, and path tracing) in the Intel® OSPRay [17, 8] rendering framework. However, our method would be applicable to other ray tracing software architectures, including NVIDIA OptiX [13].

44.2 OVERVIEW

44.2.1 RBF AND SPH FIELDS

A radial basis function is a continuous, real-valued function ϕ of distance from a center of a particle. A RBF scalar field Φ is defined by summing the kernels for all particles i contributing to a point $\mathbf{x}$ in space:

$$d_i(\mathbf{x}) = ||\mathbf{x} - \mathbf{x_i}||, \qquad \Phi(\mathbf{x}) = \sum_i \phi_i \left(d_i(\mathbf{x})\right). \tag{44.1}$$

Smoothed particle hydrodynamics are an extension of RBFs that apply a partition of unity rule.

In choosing ϕ, one can opt for kernels with either infinite support or compact support such as a polynomial. In our implementation, we use Gaussians:

$$\phi_i(x) = w_i\, e^{-\frac{1}{2}(\mathbf{d_i}(\mathbf{x})/\mathbf{r_i})^2}, \tag{44.2}$$

where w_i is the weight and r_i is the radius of the Gaussian. In practice, we truncate at a support radius of σr_i (defaulting to $\sigma = 3$); this behavior can change based on the chosen transfer function and performance needs of the user.

44.2.2 VOLUME RENDERING AND DELTA TRACKING

The Radiative Transfer Equation (RTE) [1, 9] describes how emission, extinction, and in-scattering change radiance along a ray $x(t) = o - t \cdot \omega$ inside a participating medium.

It relies on three coefficients to model these effects: the *scattering coefficient* $\sigma_s(t)$, the *absorption coefficient* $\sigma_a(t)$, and the *emission coefficient* $\sigma_e(t)$. Because extinction can be due to both absorption and out-scattering, the combined *extinction coefficient* $\sigma_t(t) = \sigma_a(t) + \sigma_s(t)$ is often used. The *phase function* is a

probability density function $\varphi(\omega_i \to \omega)$ that models which part of the incident radiance from direction ω_i is scattered into the ray.

With *transmittance* $T(0, t) = \exp(-\int_0^t \sigma_t(s)\mathrm{d}s)$, the solution to the RTE can be derived as the recursive formulation:

$$L(0, \omega) = \int_0^\infty T(0, t) \cdot \left[\sigma_e(t) L^e(t, \omega) + \sigma_s \int_{4\pi} \varphi(\omega_i \to \omega) L(t, \omega_i) \mathrm{d}\omega_i\right] \mathrm{d}t. \tag{44.3}$$

In scientific visualization, the RTE is often simplified by assuming that the medium does not scatter ($\sigma_s = 0$), instead relying purely on emission and absorption.

Equation 44.3 simplifies accordingly:

$$L(0, \omega) = \int_0^\infty T(0, t) \cdot \sigma_e(t) L^e(t, \omega) \mathrm{d}t. \tag{44.4}$$

Additionally, a *transfer function* is often used to modify L^e and T based on the underlying scalar field Φ, a color color map $\mathbf{C}$, and an opacity map α:

$$L^e(t, -\omega) = L^e\left(\mathbf{C}(\Phi(t)), -\omega\right), \qquad T(t, -\omega) = T\left(\alpha(\Phi(t)), -\omega\right). \tag{44.5}$$

Numerical integration of Equation 44.3 involves a choice of where and how frequently to sample this medium. The most straightforward approach picks a fixed distance between samples and approximates the value of the integral using quadrature. This approach is appropriate for deterministic volume rendering (i.e., ray marching), though it can lead to undersampling artifacts.

In Monte Carlo rendering (i.e., path tracing), uniform sampling invariably introduces bias. Instead, random distances to volume events are generated, usually by sampling the transmittance function. In homogeneous participating media, this can be done analytically.

A popular solution in the presence of spatially varying participating media is to use the method of Woodcock et al. [19]. The key insight of the method is that one may fill the medium with "virtual" particles such that it is essentially homogeneous, and apply the usual analytic sampling method using the new effective extinction coefficient $\sigma_{t,\max}$:

$$t = -\frac{\ln(1 - \xi)}{\sigma_{t,\max}}, \tag{44.6}$$

where $\xi \in [0, 1)$ is a uniformly distributed random number. Upon sampling an event, one must then reject interactions with "virtual" particles with

probability

$$P[\text{virtual}] = 1 - \frac{\sigma_t}{\sigma_{t,\max}}. \tag{44.7}$$

This results in a path tracing algorithm that is both adaptive and unbiased. For a more comprehensive comparison of delta tracking, we refer readers to [15].

44.3 IMPLEMENTATION

Our implementation in Intel OSPRay [8] and Open Volume Kernel Library (VKL) [7] allows for use in a wide range of rendering applications: ray casting, ray tracing with ambient occlusion rays, implicit isosurface rendering, distributed rendering with a Message Passing Interface (MPI), and path tracing. We use ISPC [6] as our kernel language; semantically, it is similar to CUDA or HLSL except that `uniform` variables are assumed to be constant across SIMD lanes or compute threads. All code is open source under the Apache 2.0 license and maintained in OSPRay and Open VKL.

44.3.1 PREPROCESS AND MAXIMUM VALUE ESTIMATION

The preprocess pipeline for a particle volume consists of building a bounding volume hierarchy around individual particles. A `radiusSupportFactor` parameter is given by the user as a multiple of the base particle radius (defaulting to 3). This determines the BVH bounds' tightness; smaller values improve performance at the cost of potential artifacts due to RBF truncation. We then build the BVH using Embree [5] builders.

For delta tracking, it is crucial to know an accurate maximum density value to determine $\sigma_{t,\max}$ in Equation 44.6. Moreover, by knowing local value ranges of the RBF field over the BVH nodes a priori, we can determine base sampling rates when stepping through the volume, using either ray casting or delta tracking. By mapping these values into a transfer function, we also know whether to skip whole regions of a volume. Estimating this value over the entire RBF field is a global optimization problem. To find these minima and maxima, we have implemented a heuristic that estimates the value ranges for all BVH leaf nodes. Our algorithm iterates over the set of leaf nodes of the BVH; currently, each leaf is built over one particle. Within each leaf node, we evaluate the RBF field Φ over a small grid (e.g., 10^3 points) and determine the local minimum and maximum value of those samples. Samples are computed using the same kernel used in rendering, given in the pseudocode in Listing 44-1. We then update the value ranges of interior nodes and store the global minimum and maximum values.

Listing 44-1. *Intersect and sample.*

```
bool intersectAndSampleParticle(VKLParticleVolume *uniform self,
        uniform uint64 id, float &result, vec3f samplePos)
{
  float value = 0.f;
  uniform vec3f particleCenter = get_vec3f(self->positions, id);
  uniform float radius   = get_float(self->radii, id);
  uniform float w = get_float(self->weights, id);
  delta = samplePos - particleCenter;

  if (length(delta) < radius * self->radiusSupportFactor) {
    value = w * expf(-0.5f * dot(delta, delta) / (radius * radius));
  }
  result += value;

  if (self->clampMaxCumulativeValue > 0.f) {
    result = min(result, self->clampMaxCumulativeValue);
    // Early termination of RBF evaluation
    return all(result == self->clampMaxCumulativeValue);
  }
  return false;
}

inline float VKLParticleVolume_sample(const Sampler *uniform sampler,
          const vec3f &samplePos)
{
  const VKLParticleVolume *uniform self =
      (const VKLParticleVolume *uniform)sampler->volume;

  float sampleResult = 0.f;
  traverseBVH(bvhRoot,
                sampler->volume,
                intersectAndSampleParticle,
                sampleResult,
                samplePos);
  return sampleResult;
}
```

Value estimation has a worst-case complexity of $O(N^3)$ and can be costly. It is sensible for small data sets with unpredictable behavior, i.e., if there is significant overlap of differently weighted particles, or if the user cannot accurately guess a cumulative maximum value. In many other cases, for example, the large cosmology data sets, it suffices for the user to manually specify `clampMaxCumulativeValue` and opt to avoid this preprocess step.

44.3.2 RBF VOLUME SAMPLING IN OPEN VKL

The core of our implementation in Open VKL is traversal of a BVH built around the particle data, with a custom intersection routine in which we evaluate a RBF at the sample position `samplePos` and accumulate the `result` value. To

do this, we call `traverseBVH()` (the equivalent of `rtTrace()` in OptiX). Currently, we use a software implementation of BVH traversal in Open VKL, but in principle this could equally leverage hardware traversal. Pseudocode is given in Listing 44-1.

Two optimizations are worth noting. The `radiusSupportFactor` is used within the kernel to explicitly clamp the value to zero outside of that chosen support radius. A smaller support radius leads to better performance but can introduce artifacts. The other parameter, `clampMaxCumulativeValue`, allows us to end evaluation of the RBF field when the accumulated `result` has already exceeded the value. This is particularly helpful in regions of significant overlap, where the user cares more about the contours of the volume data than the dynamic range.

A final advantage of Gaussian radial basis functions is that their derivatives can be computed analytically and used in shading (e.g., with a Lambertian model). This is implemented and exposed through the Open VKL API; we refer interested readers directly to that source code [7].

44.3.3 RENDERING IN OSPRAY

Given this mechanism for sampling a RBF volume, we can pair it with any volume rendering technique. For this, we use the volume rendering code abstracted to any VKL volume type, implemented in OSPRay [8] and also written in ISPC. The pseudocode in Listing 44-2 illustrates delta tracking as used to step through the volume.

Interval iterators are a standard feature of Open VKL and an optimization of note. Each node of the BVH contains a value range that can be queried at traversal time; if the data range falls outside the range of interest specified by the transfer function, we do not need to compute samples at all within that interval. The effectiveness of space-skipping in Open VKL depends on the sparsity of the particle data and the accuracy of the maximum value estimation described in Section 44.3.1.

We refer interested readers to the OSPRay source code [8] for volume ray casting and implicit isosurface renderers.

44.4 RESULTS AND CONCLUSION

Evaluating RBF path tracing performance is challenging because of the lack of reference points, particularly for our CPU-only implementation. For the

Listing 44-2. *Delta tracking.*

```
float delta_tracking(VolumetricModel *uniform vModel,
                     range1f rInterval,
                     vec3f &o,
                     vec3f &w,
                     float &sigma_t, // Extinction coefficient
                     vec3f &albedo)
{
  VKLIntervalIterator intervalIterator = vklInitIntervalIteratorV(
    vModel, o, w, rInterval, intervalIteratorBuffer);

  float t = 0.f;
  VKLInterval interval;
  while (vklIterateIntervalV(intervalIterator, &interval)) {
    t = interval.tRange.lower;

    const float maxOpacity =
     vModel->transferFunction->getMaxOpacity(interval.valueRange));

    // Estimate sigma_max based on the interval value.
    const float sigma_max = vModel->densityScale * maxOpacity;
    if (sigma_max <= 0.f)
      continue;

    while (true) {
      // Delta tracking
     float xi = RandomSampler_getFloat(randomSampler);

      const float dt = -log(1.f - xi) / sigma_max;
      t += dt;
      if (t > interval.tRange.upper)
        break;

      xi = RandomSampler_getFloat(randomSampler);
      const vec3f p = o + t * w;

      const float sample = vklComputeSampleV(
          vModel->volume->vklSampler,
          (const varying vkl_vec3f *uniform) & p);
      if (isnan(sample))
        continue;

      const vec4f color =
       vModel->transferFunction->get(vModel->transferFunction, sample);
      sigma_t = vModel->densityScale * color.w;

      // Rejection sampling
      if (xi < sigma_t / sigma_max) {
        albedo = make_vec3f(color);
        return t;
      }
    }
  }
  return inf;
}
```

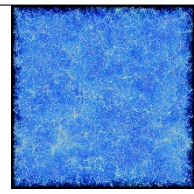
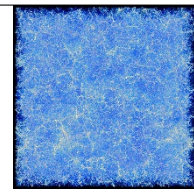
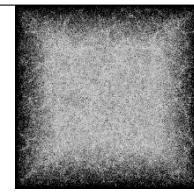
			
Modality	**Particle Volume**	**Structured Volume**	**Sphere Glyphs**
Path tracing (MR/s)	1.44	2.85	5.0
Ray casting (MR/s)	6.0	11.4	16.0
Memory usage (GB)	9.6	10.6	5.6

Table 44-1. *Performance in megarays per second (MR/s) for reference data sets (reference images shown with ray cast rendering).*

purposes of this chapter, we compared performance of ray casting and path tracing particle volumes to structured volumes, as well as rendering opaque spherical glyphs, shown in Table 44-1. The data set in question is a 51 million particle N-body cosmology code (HACC [4]).

We used a 2048^3 structured volume for comparison. Both particle and structured volume renderings required a high sampling rate of eight samples per particle radius (equivalent to four samples per voxel at 2048^3) to resolve features. Even then, striation artifacts from undersampling are clearly visible upon zooming into structured volumes. For context, we compared with opaque sphere rendering performance in Embree. We used a 2048×2048 framebuffer and one sample per particle in all cases, and reported numbers in megarays per second, running on 28 cores of an Intel Xeon 8180M at 2.5 GHz. In all, performance with both path tracing and ray casting is interactive, though not in real time.

Some room for improvement remains. GPU implementations, including ones that leverage GPU ray tracing hardware, would be of interest and should be a straightforward extension. Improving memory efficiency is also important. We currently build the BVH around individual particles as opposed to clustering, resulting in a roughly $5\times$ overhead relative to original (float) particle data. To handle larger particle visualizations, we could pursue compression or clustering methods as explored in [2].

In conclusion, the advantage of RBF volumes lies in quality: we can inspect individual particle elements from the simulation if needed and know that nothing is omitted or simplified. Delta tracking, used in conjunction with

interval iteration based on local value ranges, allows for efficient and unbiased path tracing. Although specialized ray casting approaches (e.g., [10]) would likely deliver superior performance, the versatility of being able to path-trace and operate within an abstracted volume sampling interface (Open VKL) is useful.

REFERENCES

[1] Chandrasekhar, S. *Radiative Transfer*. Oxford, 1960.

[2] Gralka, P., Wald, I., Geringer, S., Reina, G., and Ertl, T. Spatial partitioning strategies for memory-efficient ray tracing of particles. In *2020 IEEE 10th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 42–52, 2020.

[3] Green, S. Volumetric particle shadows. *NVIDIA Developer Downloads*, https://developer.download.nvidia.com/assets/cuda/files/smokeParticles.pdf, 2008.

[4] Habib, S., Pope, A., Finkel, H., Frontiere, N., Heitmann, K., Daniel, D., Fasel, P., Morozov, V., Zagaris, G., Peterka, T., et al. HACC: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy*, 42:49–65, 2016.

[5] Intel. Intel Embree: High preformance ray tracing kernels. https://www.embree.org, 2020.

[6] Intel. Intel implicit SPMD program compiler. https://ispc.github.io.

[7] Intel. Intel Open Volume Kernel Library. https://www.openvkl.org, 2021.

[8] Intel. Intel OSPRay: The open, scalable, and portable ray tracing engine. https://www.ospray.org, 2021.

[9] Kajiya, J. T. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, pages 143–150, 1986.

[10] Knoll, A., Morley, R. K., Wald, I., Leaf, N., and Messmer, P. Efficient particle volume splatting in a ray tracer. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 533–541. Apress, 2019.

[11] Knoll, A., Wald, I., Navratil, P., Bowen, A., Reda, K., Papka, M. E., and Gaither, K. RBF volume ray casting on multicore and manycore CPUs. *Computer Graphics Forum*, 33(3):71–80, 2014.

[12] Lorensen, W. E. and Cline, H. E. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pages 163–169, 1987.

[13] NVIDIA. NVIDIA OptiX ray tracing engine. https://developer.nvidia.com/optix, 2021.

[14] Orthmann, J., Keller, M., and Kolb, A. Topology-caching for dynamic particle volume raycasting. In *Vision, Modeling, and Visualization 2010*, pages 147–154, 2010.

[15] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 3rd edition, 2016.

[16] Raab, M. Ray tracing inhomogeneous volumes. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 521–531. Apress, 2019.

[17] Wald, I., Johnson, G. P., Amstutz, J., Brownlee, C., Knoll, A., Jeffers, J., Günther, J., and Navrátil, P. OSPRay—A CPU ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):931–940, 2016.

[18] Westover, L. Footprint evaluation for volume rendering. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, pages 367–376, 1990.

[19] Woodcock, E, Murphy, T, Hemmings, P, and Longworth, S. Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry. In *Proceedings Conference Applications of Computing Methods to Reactor Problems*, page 557, 1965.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 45

FAST VOLUMETRIC GRADIENT SHADING APPROXIMATIONS FOR SCIENTIFIC RAY TRACING

Carson Brownlee and David DeMarle
Intel Corporation

ABSTRACT

In scientific visualization, direct volume rendering requirements range from fast approximations for interactive data exploration to photorealism with physically based shading for renderings destined for the general public. Gradient shaded lighting is a common non-physically based shading method used in scientific volume visualization that increases the level of realism and helps distinguish features in the data over simple forward ray marching. Gradient shading comes at a significant runtime cost over simple ray casting, especially as secondary rays are calculated at each sample point for lighting effects such as shadows and ambient occlusion. In this chapter we examine the simple optimization of shading at the single point of highest contribution. In practice this method gives most of the benefits of full gradient shading that help in feature detection and realism, with an order of magnitude performance improvement.

45.1 INTRODUCTION

Direct volume rendering has been a staple of scientific visualization of three-dimensional scalar fields since the 1980s [2]. Advances in rendering algorithms and hardware acceleration have enabled an offline process to be computed interactively, and then in real time, even for large-scale volumes [4]. Direct volume rendering computes a piecewise-linear approximation of the volume rendering integral, calculating color attenuation and opacity based on extinction coefficients along a viewing ray $x(\lambda)$ at distance λ:

$$I = \int_0^D c\left(s(x(\lambda))\right) e^{-\int_0^\lambda \alpha(s(x(\lambda')))d\lambda'} d\lambda. \tag{45.1}$$

The color c is calculated based on a transfer function at the scalar value s,

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_45

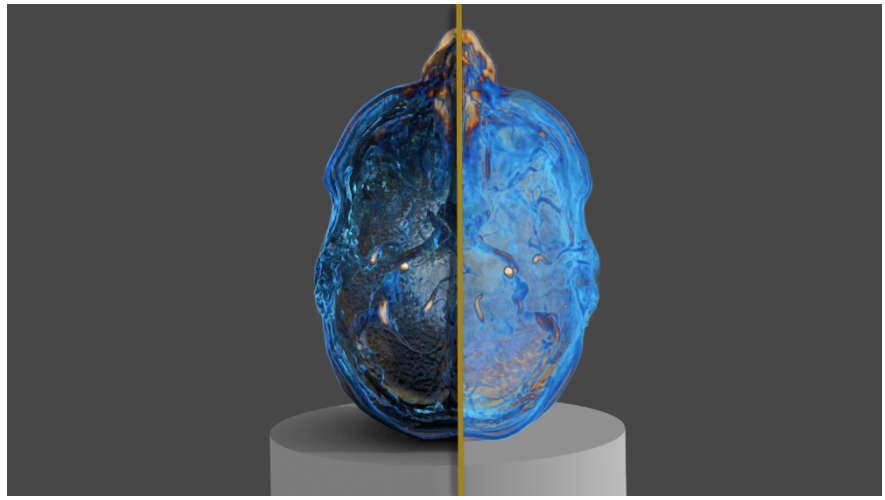


Figure 45-1. *A scan of a human head rendered with simple volume ray casting (right) and with shadows and ambient occlusion (left). Reducing the number of shading samples enables this common scientific rendering method to be done at interactive to real-time rates.*

$c(s(x(\lambda)))$. Similarly, the opacity α is calculated based on a transfer function based on the scalar value, $\alpha(s(x(\lambda)))$.

To show surface illumination at each point, c can be modified to use lighting information for shading based on the gradient of the scalar field at point x, $\nabla s(x)$. Figure 45-1 shows direct volume rendering with simple line of sight ray marching on the right and the same image but with shading, ambient occlusion, and shadows on the left. The piecewise-linear approximation of the integral is shown in illustrated form in Figure 45-2. Figure 45-1 shows images produced from the integration shown in Figure 45-2a for ray casting and Figure 45-2b for ray tracing with secondary rays. For ray casting, this is an

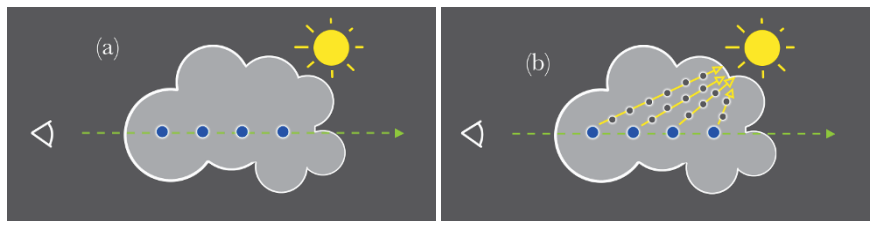


Figure 45-2. *(a) Direct volume rendering using line of sight ray marching. (b) Ray casting shadow rays calculated at every sample point along the eye ray.*

$O(n)$ computation per viewing ray, for n volume samples. Gradient shading adds multiple additional volume lookups at each point to compute $\nabla s(x)$, while adding a single shadow ray or ambient occlusion sample at each point makes it an $O(n^2)$ computation for each viewing ray. We refer to the full gradient shading method as *FG* and a faster approximation using a single shading point as single shot gradient shading, *SSG*.

These approaches are used in the Visualization Toolkit (VTK) [7]. VTK is the backbone of many of the most commonly used open source visualization programs such as ParaView [8] and VisIt [1]. VTK has two primary ray traced rendering options in Intel OSPRay [9] and NVIDIA OptiX [6], and all of VTK's volume renderers support gradient shading. ParaView has supported built-in ray tracing of volumes via OSPRay 1.x since version 5.2, including support for SSG shading up through 5.8. Note that in ParaView 5.9, with the introduction of major OSPRay 2.x features including path traced volume rendering, ParaView temporarily lost SSG rendering because it was not part of OSPRay 2's initial feature set due to a number of feature regressions with the SciVis renderer. This feature is being reintroduced in OSPRay 2.6 and should reappear in ParaView 5.10. ParaView 5.9 with a OSPRay 2.6 pre-release is used here for all results and images.

45.2 APPROACH

The approach utilized is very simple in form and implementation: calculate an average of all gradients weighted by alpha contribution, then calculate shading and secondary rays from the single point of highest contribution, P_{max}, using the averaged gradient. Alternatively, the highest contributing gradient could be used, but averaging minimizes view-dependent changes in the resulting gradient. Pseudocode for both full gradient shading and using a single shading point, determined with the `useSingleShadeHeuristic` variable, is as follows:

```
void TraceVolumeRay()
{
  float4 finalColor = {0,0,0,0};
  while(!steppingDone) {
    // Compute color and opacity at current sample point.
    sample = Volume_getValue(distance);
    color = mapThroughLookupTable(sample);
    alpha = (1.f - finalAlpha) *
      (1.f - exp(-color.a * dt * densityScale));

    // Compute gradient shading lighting.
    float4 ns = Volume_getGradient(distance);
    if (useSingleShadeHeuristic) {
```

```
      // Remember point of highest density for deferred shading.
      if (highestContribution < alpha) {
        highestContribution = alpha;
        highestContributionDistance = distance;
      }
      // Accumulate alpha and gradient.
      finalAlpha += alpha;
      gradient += ns * alpha;
    } else {
      // At every point, cast rays for contributions to integrate.
      shading = computeShading(distance, ns);
      color = lerp(gradientShadingScale, color, shading);
    }
    finalColor.blend(color, alpha);
    steppingDone = increase(&distance);
  }

  if (useSingleShadeHeuristic) {
    // At a single, point cast rays for contribution.
    float3 ns = normalize(gradient);
    shading = computeShading(highestContribtionDistance, ns);
    singleShading = shading * finalAlpha;
    finalColor = lerp(gradientShadingScale, finalColor, singleShading);
  }
}
```

Alternate methods for reducing the number of samples required include using isosurfaces (meshes or implicit) or utilizing methods such as delta tracking progressively over many passes. Isosurfaces at peaks of the transfer function minimize shading operations, however the transfer function corresponding to a set of isosurfaces consists of a set of impulses in opacity. The more a user-defined transfer function deviates from this form, the less the resulting image would match the base gradient shaded image. With regards to delta tracking, OSPRay's path tracing algorithm utilizes Woodcock style delta tracking [5], which reduces bias in sampling volumes compared to fixed-step ray marching. ParaView's OSPRay-based path tracing thus uses the same implementation. The images and performance numbers shown in Section 45.3.2 demonstrate the core problem we solve, that too many passes are still needed to achieve a noise free image. Denoising reduces the number of samples needed, but introduces its own overhead and unacceptable levels of feature blurring at lower sample counts.

45.3 RESULTS

All runs are performed on a dual socket Intel Xeon Gold 5220, with 36 total cores, 256 GB of RAM, and a GeForce RTX 2060. The GPU is used only for display in these results as OSPRay is currently CPU-only. An unmodified version of ParaView 5.9.0 is used with a pre-release version of OSPRay 2.6

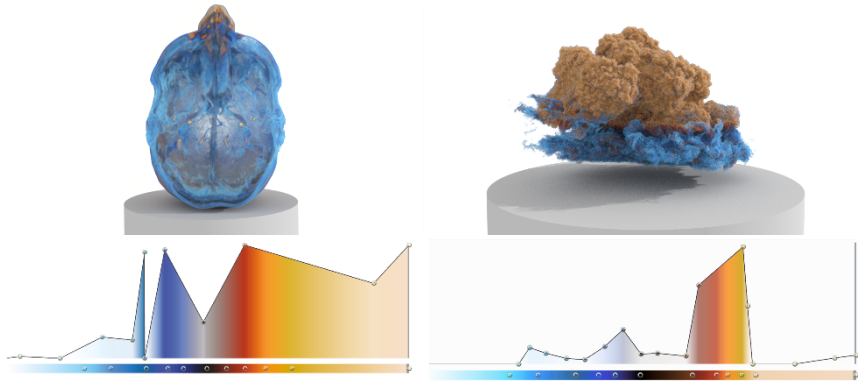


Figure 45-3. *Brain (left) and cloud (right) data sets with respective transfer functions.*

that supports single shading. The data sets used are an example scan of a human head with a tumor from the 3D Slicer [3] project and the Disney cloud [10], shown in Figure 45-3. The color and opacity transfer functions are shown below their respective renderings, with opacity relating to the height of the control points over the transfer function range. For the following comparisons, we will relate three different rendering methods:

- *Full gradient (FG):* Gradient shading where each eye ray sample point computes a gradient and secondary rays at each sample point.
- *Single shot gradient (SSG):* Gradient shading where only a single sampling point along the ray is used for gradient shading operations.
- *Path traced (PT):* Path tracing using Woodcock-style delta tracking. We include this as a another physically based method of shading based on volume gradients increasingly used in scientific visualization and now standard in studio rendering.

45.3.1 IMAGE QUALITY COMPARISON

Figures 45-4 and 45-5 show path traced images of the cloud and brain data sets. The first image with a single sample per pixel (spp) is very noisy. At 4 spp the denoiser is enabled, which reduces noise, but the image becomes very blurry. Though the image starts to look significantly better, we feel that the blurred-over features of the data are likely unacceptable for scientific visualization applications until higher sample counts are used at around 32 spp. PT with 32 spp is faster than 32 ambient occlusion (AO) samples with

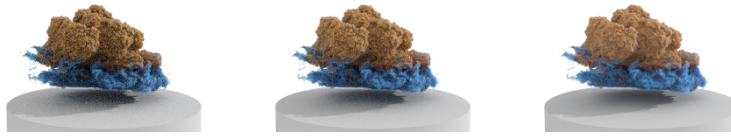


Figure 45-4. *Cloud path traced. From left to right: 1 spp, 4 spp (denoised), and 32 spp (denoised).*

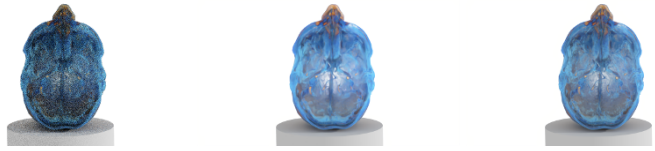


Figure 45-5. *Brain path traced. From left to right: 1 spp, 4 spp (denoised), and 32 spp (denoised).*

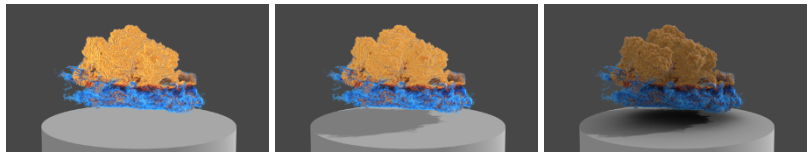


Figure 45-6. *Cloud ray traced with gradient shading using FG. From left to right: ray casting, shadows, and shadows and AO.*

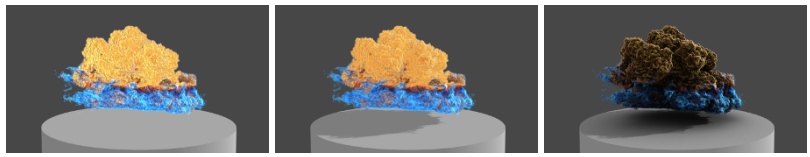


Figure 45-7. *Cloud ray traced with SSG. From left to right: ray casting, shadows, and shadows and AO. SSG produces lighter highlights and darker shadows.*

FG due to the increased number of samples for the AO rays using the FG method compared to the stochastic sampling of secondary rays using delta tracking with PT.

Figures 45-6 and 45-7 show renderings of the cloud data sets with FG and SSG shading, respectively. SSG shading exhibits higher contrast in shadows and highlights as the shaded color is not averaged over multiple samples.

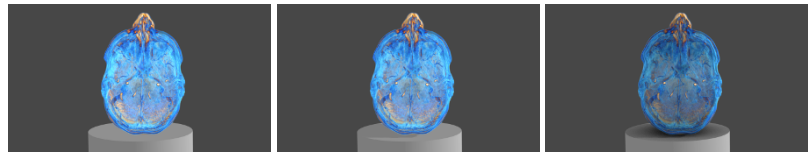


Figure 45-8. *Brain ray traced with gradient shading with FG. From left to right: ray casting, shadows, and shadows and AO.*

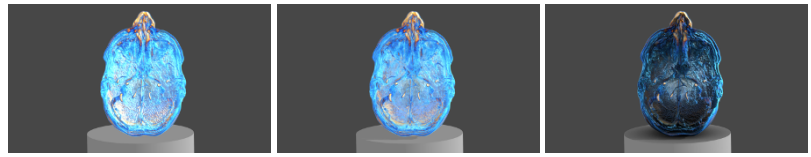


Figure 45-9. *Brain ray traced with SSG. From left to right: ray casting, shadows, and shadows and AO.*

This becomes even more apparent as shadow and ambient occlusion rays are added, where the highest contributing point tends to be deeper inside the volume. Pixel parity with FG shading is not paramount in a scientific visualization context as the primary aim is enhanced identification of areas of curvature and proximity, though FG is not a physically based rendering algorithm to begin with. SSG also has a drawback of view-dependent shading points as the point of greatest contribution changes based on viewing angle; however, in practice no users have reported differences between the two and have been happy with the added fidelity of shadows and ambient occlusions introduced in ParaView. Figures 45-8 and 45-9 show the same comparisons using the brain data set. Here, the most striking difference is when ambient occlusion is added. SSG in this instance significantly intensifies occlusion inside the skull surface, enhancing the bone surface.

45.3.2 PERFORMANCE COMPARISON

Table 45-1 shows recorded performance numbers for various methods rendered at 1080p resolution inside of ParaView. Performance increases for single shading varies between ca. 5× to ca. 50× for a single shadow ray to a shadow ray and 32 ambient occlusion samples. Though dramatically different methods, the single shade method is significantly faster than delta tracking with a single sample per pixel. The 1 spp delta tracking method does not use

Ray Tracing (s)				
Model	Ray Cast	Shadows	Shadows & 1 AO	Shadows & 32 AO
Brain FG	0.102	0.486	1.25	24.3
Brain SSG	0.091	0.097	0.142	0.454
Cloud FG	0.329	1.07	3.62	73.9
Cloud SSG	0.180	0.207	0.265	1.95
Path Tracing (s)				
Model	1 spp	4 spp (denoised)	32 spp (denoised)	
Cloud PT	0.643	2.44	19.0	
Brain PT	0.401	1.49	11.7	

Table 45-1. *Performance numbers for the brain and cloud models using FG, SSG, and PT. Ray casting computes only direct illumination, shadows adds a single shadow ray, shadows and 1 AO adds one shadow and one ambient occlusion ray, and shadows and 32 AO adds a single shadow ray and 32 ambient occlusion rays.*

a denoiser, while the denoiser is used for 4 spp and 32 spp. In the current use, we feel that the path traced image is not sufficiently crisp until at least 32 spp, with lower sample counts making features in the data difficult to discern.

45.4 CONCLUSION

Single shot gradient shading provides an order of magnitude faster approximation for scientific visualization that, in practice, is a ready stand-in for full gradient shading. Interactive rates are maintained for high sample counts without the need for many passes and denoising. The method introduces error compared to the full gradient shading baseline, but given the non-physically based nature of gradient shading, this does not seem to be a significant issue in practice. This simple method was vital to adding secondary rays to basic scientific direct volume rendering in VTK.

REFERENCES

[1] Childs, H. et al. VisIt: An end-user tool for visualizing and analyzing very large data. https://escholarship.org/uc/item/69r5m58v, 2012.

[2] Engel, K., Kraus, M., and Ertl, T. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 9–16, 2001. DOI: 10.1145/383507.383515.

[3] Fedorov, A., Beichel, R., Kalpathy-Cramer, J., Finet, J., Fillion-Robin, J.-C., Pujol, S., Bauer, C., Jennings, D., Fennessy, F., Sonka, M., Buatti, J., Aylward, S., Miller, J. V., Pieper, S., and Kikinis, R. 3D slicer as an image computing platform for the Quantitative Imaging Network. *Magnetic Resonance Imaging*, 30(9):1323–1341, 2012. DOI: 10.1016/j.mri.2012.05.001.

[4] Meißner, M., Hoffmann, U., and Straßer, W. Enabling classification and shading for 3D texture mapping based volume rendering using OpenGL and extensions. In *Proceedings of the Conference on Visualization '99: Celebrating Ten Years*, pages 207–214, 1999.

[5] Morgan, L. and Kotlyar, D. Weighted-delta-tracking for Monte Carlo particle transport. *Annals of Nuclear Energy*, 85:1184–1188, 2015. DOI: 10.1016/j.anucene.2015.07.038.

[6] NVIDIA. NVIDIA OptiX ray tracing engine. https://developer.nvidia.com/optix, 2021. Accessed February 8, 2021.

[7] Schroeder, W. J., Martin, K. M., and Lorensen, W. E. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *Proceedings of Seventh Annual IEEE Visualization '96*, pages 93–100, 1996. DOI: 10.1109/VISUAL.1996.567752.

[8] Squillacote, A. H., Ahrens, J., Law, C., Geveci, B., Moreland, K., and King, B. *The ParaView Guide*. Kitware, 2007.

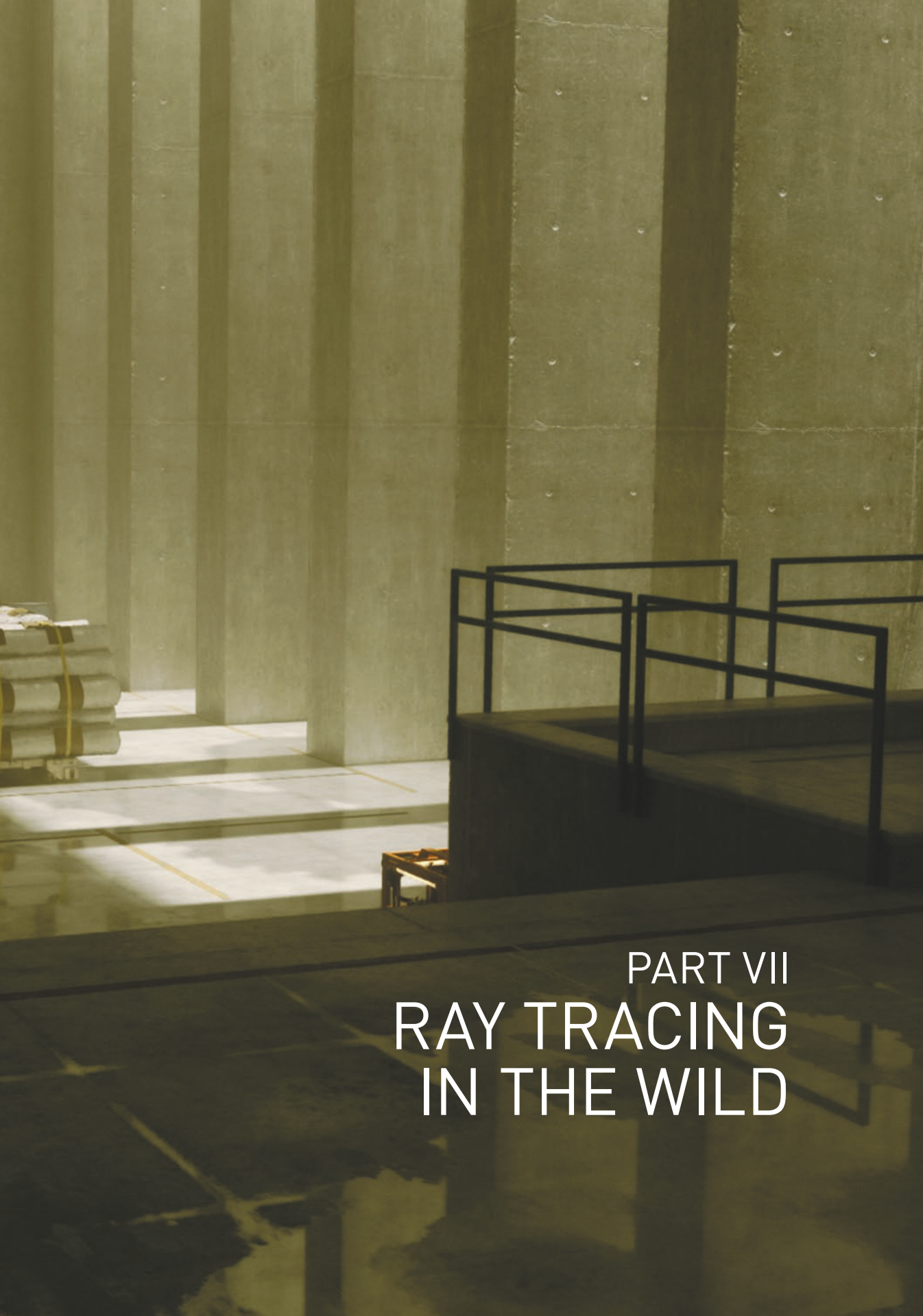
[9] Wald, I., Johnson, G., Amstutz, J., Brownlee, C., Knoll, A., Jeffers, J., Günther, J., and Navratil, P. OSPRay—A CPU ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):931–940, 2017. DOI: 10.1109/TVCG.2016.2599041.

[10] Walt Disney Animation Studios. Data sets: Clouds. https://www.disneyanimation.com/data-sets/?drawer=/resources/clouds/, 2020. Accessed February 8, 2021.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





PART VII
RAY TRACING IN THE WILD

PART VII

RAY TRACING IN THE WILD

With modern GPUs, ray traced effects are now feasible in real-time graphics pipelines. This provides an opportunity for higher visual quality, as ray tracing can more easily capture realistic illumination, including reflections, refractions, shadows, and global illumination. However, there is no free lunch. Game engines have been optimized for rasterization-based graphics pipelines over many years, and integrating ray tracing requires careful design considerations. Furthermore, high-quality ray traced images use thousands of rays per pixel at high resolutions, which is not yet feasible in real time. This implies that ray tracing with a limited ray budget must be coupled with denoising techniques, in order to meet both the performance and quality targets.

Despite these challenges, there has been tremendous progress in ray tracing adoption in games. Ray tracing is now readily available in some of the most popular game titles and sets a new bar for visual fidelity in interactive applications. This part presents examples of how this has been accomplished.

Control was one of the first games with ray tracing. Chapter 46, *Ray Tracing in Control*, describes the game's implementation of ray traced reflections, contact shadows, indirect diffuse illumination, and corresponding denoisers. The visual results, generated in real time, are stunning.

The version of *Quake 2* enhanced by ray tracing (the *Quake 2* RTX project, Q2VKPT) is a nice showcase of applying path tracing to a classic game title. Chapter 47, *Light Sampling in Quake 2 Using Subset Importance Sampling*, addresses a highly active research topic: how to efficiently ray-trace scenes with a large number of light sources. Random selection of subsets of lights per pixel is combined with multiple importance sampling within each subset.

At the time of writing, *Fortnite* is the most popular game in the world. Chapter 48, *Ray Tracing in Fortnite*, explains the integration of ray tracing techniques, including shadows, reflections, and global illumination, into *Fortnite* Season 15. By combining carefully tuned ray traced effects with modern denoising techniques and NVIDIA's Deep Learning Super Sampling (DLSS), the team was able to substantially increase visual quality while also meeting the game's strict performance target.

Chapter 49, *ReBLUR: A Hierarchical Recurrent Denoiser*, describes a novel denoiser designed for real-time ray tracing applications. It has been deployed in recent titles, including *Cyberpunk 2077* and *Watch Dogs: Legion*. By aggressively reusing information both spatially and temporally, the denoiser produces practically useful results from only a few samples per pixel.

Chapter 50, *Practical Solutions for Ray Tracing Content Compatibility in Unreal Engine 4*, addresses challenges and solutions when working with ray tracing in scenarios where applications must also support legacy hardware. Two techniques are covered: hybrid translucency and animated foliage, both of which have been successfully used in shipping games.

These chapters present multiple examples of successful applications of ray tracing in recent game titles. Alongside beautiful visual results, the challenges and techniques to make them possible are carefully discussed. I hope these chapters serve as inspiration for future ray traced interactive applications.

Jacob Munkberg

CHAPTER 46

RAY TRACING IN CONTROL

Juha Sjöholm,[1] Paula Jukarainen,[1] and Tatu Aalto[2]
[1]NVIDIA
[2]Remedy Entertainment

ABSTRACT

In this chapter, we describe how ray tracing was used in *Control*. We explain how all ray tracing–based effects, including opaque and transparent reflections, near field indirect diffuse illumination, and contact shadows, were implemented. Furthermore, we describe the denoisers tailored for these effects. These effects resulted in exceptional visual quality in the game, while maintaining real-time frame rates.

46.1 INTRODUCTION

Control, launched in 2019, was one of the very first games with ray tracing. Here, ray tracing was utilized in multiple ways to achieve higher visual quality. *Control* uses a hybrid rendering approach, combining rasterization and ray



Figure 46-1. *Control uses ray traced effects, such as reflections and near field indirect diffuse illumination, to add to its unique artistic style. (Image courtesy of DeadEndThrills.)*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_46

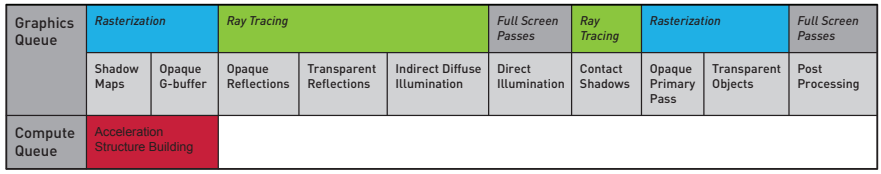


Figure 46-2. *A simplified breakdown of a frame in Control with ray tracing effects enabled showing how both rasterization and ray tracing are used for different purposes.*

tracing. Ray tracing is used in opaque and transparent reflections, near field indirect diffuse illumination, and contact shadows. These effects combined demonstrate that ray tracing can achieve a new level of realism in real-time gaming. A simplified breakdown of a frame can be seen in Figure 46-2.

This introduction section explains common features of the game engine utilized by the different ray tracing effects. The rest of the chapter goes into the details of how each effect, including denoising, was implemented in *Control*. We describe how each effect was optimized to maintain real-time frame rates. We highlight two recurring strategies that were the keys to meeting the performance and quality targets:

- Reducing incoherence by shortening rays when possible.
- Shading incoherent rays at the right level of accuracy, which both reduces noise and improves performance.

46.1.1 NORTHLIGHT ENGINE

Control was developed using the Northlight engine, an in-house game engine developed by Remedy Entertainment. The Northlight engine uses deferred lighting with a bindless material system, which simplifies the implementation and optimization of ray tracing effects, e.g., effects that trace and shade secondary rays (discussed in Sections 46.2, 46.3, and 46.4). It also allows tracing shadow rays for selected light sources for each pixel (discussed in Section 46.5). Additionally, the engine supports an approximative unified parameterization and shading model for all materials, which is used in shading of the ray hits. That works especially well with incoherent rays, as described in Section 46.2.2.

46.1.2 PRECOMPUTED GLOBAL ILLUMINATION

The Northlight engine supports precomputed voxel-based global illumination (see Aalto [1] for details), which had a key role in optimizing the ray traced

reflections and indirect diffuse illumination (as described in Section 46.2.3 and 46.4). The precomputation is performed by a path tracer and is based on static objects and selected light sources. The game levels have the global illumination (GI) data stored in sparse volume textures. The resolution of the available data at a given location is fundamentally artist authored. The data can be sampled with a world-space position and a direction vector. The sampling result is the irradiance over the hemisphere facing the given direction in the given position.

46.1.3 ACCELERATION DATA STRUCTURE BUILDING

All ray tracing passes in the game use the same acceleration data structure. An important principle in the construction of the acceleration data structure is to use the same geometry levels of detail (LODs) as are used in rasterization. This helps with avoiding self-intersections while providing as much detail as possible for ray tracing. A design goal for the ray tracing effects was to provide more accurate details than what is possible with screen space–based techniques executed after rasterizing the scene.

For selecting the objects to be included in the ray tracing acceleration structure, an expanded camera frustum–based culling is applied to the scene objects in order to gather the objects that potentially contribute to some effect. All opaque and most alpha tested objects are included. Some alpha tested vegetation assets are left out as including them would give only minor visual benefits compared to the increased ray tracing costs they incur. Particles that are rendered as opaque meshes are also included in the acceleration data structure. Blended objects and particles are excluded, but that doesn't lead to significant visual issues. However, to support discovering transparent surfaces for rendering reflections on them, blended objects are inserted into the structure with a special cull mask. This is discussed in more detail in Section 46.3.1.

To reduce the memory and cache traffic during ray tracing, the compaction operation available in the DirectX Raytracing (DXR) API is performed on all static acceleration data structures. Skinned meshes are represented as triangle geometries by outputting the skinned vertices to buffers with a compute shader pass. The acceleration structure rebuilds and updates are modulated to achieve better overall performance, but the rasterized and ray traced meshes match on every frame. All acceleration structure building work is executed on the asynchronous compute queue, as shown in Figure 46-2.

46.1.4 LIGHT CLUSTERING

To shade specular reflections or indirect diffuse hits, we require knowledge about which lights affect a given hit location. Evaluating illumination from all scene lights would not be possible simply because of the evaluation cost. The game levels may contain several thousand dynamic lights. For rasterization, effective screen tile–based light culling implementation already existed, but that was not directly suitable for shading ray hits. For ray tracing effects, an additional light clustering pass is executed. It culls the scene lights against cells of an axis-aligned 3D grid in view space. The grid has a limited size that matches the range of the ray tracing effects. The clustering pass stores indices of the lights affecting each grid cell to a texture. When shading the ray hits, the list of lights affecting the grid cell matching the hit position is processed.

46.2 REFLECTIONS

The implementation of ray traced specular reflections in *Control* is straightforward. The reflection rays are generated for each pixel based on the view direction and the surface properties stored in the rasterized G-buffer. Self-intersections are avoided by matching the geometry LODs in ray tracing and rasterization. Additionally, due to the inaccuracy of reconstructing world-space position from the rasterized depth buffer, a bias value scaled by pixel depth toward the camera position and along the surface normal is added to the ray origin. One ray is traced for each pixel excluding only the sky pixels. The game uses the GGX specular bidirectional reflectance distribution function (BRDF) and has surfaces with spatially varying roughness levels. An important design goal was to make the reflections work consistently across all game content. Figure 46-3 shows the reflections on different surfaces.

The following sections describe the techniques that are used to find a good balance between the desired visual quality and performance. Setting the ray length, the fallback solution for missed rays, and shading quality of hits proved to be essential issues. Figure 46-4 illustrates the general workflow.

46.2.1 TRACING REFLECTION RAYS WITH VARYING RAY LENGTH

The ray direction for each pixel is importance-sampled from the GGX distribution. Higher surface roughness means that the ray directions on neighboring pixels are more incoherent, which leads to more noise in the rendered image and higher computation cost. To avoid generating noise that



Figure 46-3. *Left: screen-space reflections. Right: ray traced reflections, which show more accurate details. The denoising process explained in Section 46.6 is applied to the image.*



Figure 46-4. *A general overview of the ray traced reflections rendering.*

would need to be removed from the final image anyway in the denoising passes and also to reduce the cost of the evaluation, the length of the reflection ray is limited based on the surface roughness, as illustrated in Figure 46-5. On surfaces with the maximum roughness value 1.0, the ray length is only about 3 meters. It increases exponentially to about 200 meters as roughness decreases to the minimum value 0.0. These limits were chosen by experimentation to make the result visually plausible. To make reflected objects appear smoothly into the image as they move closer to the reflecting surface, a pixel index–based random variation is also applied to the ray length. This hides visual artifacts from the switch from ray misses to ray hits.

46.2.2 UNIFIED HIT SHADING

For shading the G-buffer, *Control* has a number of material variations with special shading models for, e.g., character skin, eyes, and hair. However, for shading the ray hits in ray tracing effects, a unified variant based on simple parameterization of physically based shading is used for all materials, as



Figure 46-5. *How the reflection ray length varies based on surface roughness. Left: longer reflection rays are generated when the roughness is low and ray direction distribution is coherent. Right: shorter rays are used when the roughness is high and the direction distribution is incoherent.*

Listing 46-1. *Pseudocode overview of the single any-hit shader used to perform alpha testing.*

```
uint material = GetHitMaterialID();
uint3 vertexIndices = GetHitVertexIndices();
float2 uv = InterpolateHitUV(barycentrics, vertexIndices);
float alpha = SampleMaterialAlpha(material, uv);

if (alpha < 0.5f)
    IgnoreHit();
```

mentioned in Section 46.1.1; i.e., the single parameterization and shading model is used for all materials. This allows *Control* to use only one any-hit shader and only one closest-hit shader in the DXR API. Overview of the unified any-hit shader can be seen in Listing 46-1. The unified path is only an approximation, but it provides a visually plausible result in practice. It makes ray tracing development easier in general and helps to achieve satisfying performance especially with incoherent rays and alpha testing. With the incoherent rays, it also reduces noise in the output. The result of the hit shading is the radiance arriving at the G-buffer surface. This is denoised, as described in Section 46.6, before applying it to the receiving surface.

46.2.3 PRECOMPUTED GLOBAL ILLUMINATION FOR RAY MISSES

When the specular reflection rays miss, the precomputed GI data (Section 46.4) is used to approximate the radiance coming from the direction of the ray. The data is sampled at the end of the missed ray. As illustrated in Figure 46-6, the irradiance over the hemisphere provided as the sampling result is converted to average radiance before using it as an approximation. Obviously, this is not accurate for multiple reasons. The result is based only on static geometry and lights, the data resolution is limited, and converting

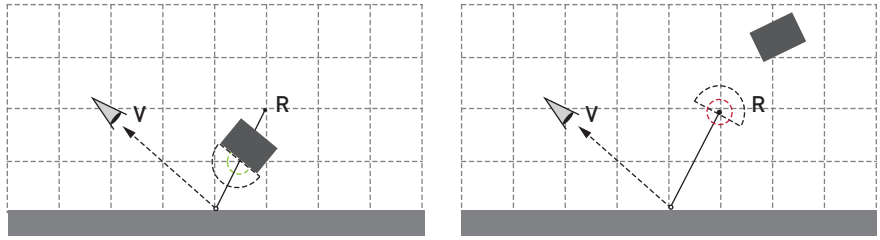


Figure 46-6. *How the precomputed GI data is used for missed rays. Left: the reflection ray hits an object and the hit is shaded. Right: the reflection ray misses the object that is too far away, and the GI data is sampled at the end of the ray as an approximation for the incoming radiance. Sampling the GI data gives irradiance over hemisphere. That is converted into average radiance.*

irradiance to average radiance can cause light leaks. But despite the limitations, this method provides a visually plausible result in practice.

46.2.4 UNIFIED GLOBAL ILLUMINATION SAMPLING FOR HITS AND MISSES

To further unify the shading process, the handling of hits and misses uses partially the same code path. This is possible as the GI data is used to approximate the second ray bounce for hits and the first bounce for misses. For hits, the GI data is sampled directly at the hit location for the second bounce approximation. This means that the hit and miss handling are not done inside hit or miss shaders. In *Control*, the shading happens in a separate compute pass dispatched after the actual ray tracing pass is completed. The separate pass reduces cache pressure especially when the rays are incoherent. It's implemented by resolving the hit geometry normal and texture coordinates in the hit shader and storing them to textures for the shading pass in addition to the hit position and material identifier. The hit position is stored in view space to make half-precision floating-point values sufficient for holding it. The ray misses are marked with a special material identifier. An overview of the compute shader with the unified GI sampling for hits and misses can be seen in Listing 46-2.

46.3 TRANSPARENT REFLECTIONS

Many levels in the game contain a fair amount glass windows, interior walls, and other items with glass surfaces, e.g., wall clocks or poster frames. Having ray traced reflections on those felt like a good addition to the reflections on opaque surfaces. The reflections in different situations are

Listing 46-2. *Pseudocode overview of the shading pass performing the GI sampling for both hits and misses.*

```
float originDepth = DecodeGBufferDepth();
float3 position = DecodeHitOrMissPosition();
uint material = DecodeHitMaterialID();
float3 normal = DecodeHitNormal();
float2 uv = DecodeHitUV();
bool isHit = isMaterialHit(material);

float3 rayDirection = ReconstructRayDirection(originDepth, position);
float3 irradiance = SampleGI(isHit, position, normal, rayDirection);
float3 radiance;

if (isHit) {
    radiance = ShadeHit(position, normal, rayDirection, material, uv,
        irradiance);
}
else {
    radiance = ConvertToAverageRadiance(irradiance);
}

WriteOutput(radiance);
```

shown in Figure 46-7. The following sections describe how the transparent surfaces that receive ray traced reflections are identified, how the reflection rays are generated, and how the results are applied to the receiving surface. Figure 46-8 illustrates the general workflow.



Figure 46-7. *Left: environment map–based reflections on transparent surfaces. Right: ray traced reflections, which show significantly more accurate details.*



Figure 46-8. *A general overview of the ray traced transparent reflections rendering.*

46.3.1 DISCOVERING TRANSPARENT SURFACES

Control uses ray tracing to find transparent surfaces. Primary rays are traced against only the transparent objects. The length of the rays is limited based on the rasterized opaque depth buffer to discover only the visible surfaces, as illustrated in Figure 46-9. If a primary ray hits a transparent surface, a reflection ray is generated based on the surface properties. Otherwise, the pixel is marked as not having a transparent surface.

An alternative to the primary rays approach could have been rasterizing a transparent G-buffer and tracing secondary rays based on that. However, performance of the ray tracing approach proved to be competitive. The directions of the primary rays are naturally coherent, and the processing applied to them is uniform and quite simple in this case.

Transparent objects are inserted into the same acceleration structure as everything else but marked with a different cull mask. Storing them in a separate structure was also tried, but because there isn't a significant overall performance difference between the two approaches in this case, using a common acceleration structure was chosen for simplicity. The overall performance is a combination of the acceleration structure build cost and the ray tracing cost.

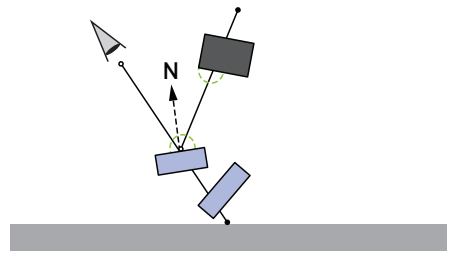


Figure 46-9. *First, discover the transparent surface by tracing a primary ray against transparent objects, and then trace a recursive reflection ray for the closest found surface. The length of the primary ray is limited based on the rasterized opaque depth.*

After discovering a transparent surface, it would be possible to continue the primary ray in order to discover the next transparent surface. However, due to performance reasons, the transparent reflections are limited only to the nearest surface. The cost of the additional reflection rays would grow too high to support more layers in some scenarios.

46.3.2 TRACING TRANSPARENT REFLECTION RAYS

Reflection rays are generated based on what the primary ray hits. One ray per pixel is used as with opaque reflections. The surface normal is evaluated, and the normal map is also applied. However, the possible surface roughness value is ignored, and the reflection ray direction is always evaluated as a perfect mirror reflection without any randomization. This allows avoiding another denoising pass. Most transparent surfaces in the game are actually mirror reflectors, so visually this works without disturbing issues.

As automatic texture LOD level selection is not available in ray tracing, evaluating an approximation for the LOD for sampling the normal map is required. Otherwise, the normal mapped reflection ray directions would be very noisy in some situations. A simple LOD evaluation based on the pixel size in world space proved to be sufficient in this case.

The length of the reflection ray is limited to 60 meters. The limit is not often reached in the game, which contains mostly indoor locations, but it is still applied to keep the performance stable under all scenarios. After the distance limit, the reflections are based on the same environment cube maps that are used when the ray traced reflections are disabled. Near the distance limit, the result is faded from the ray traced result to the cube map–based result to avoid sudden flips between different visual looks.

The shading of the reflection hits is done in the same way as for reflections on opaque surfaces. A separate compute shader pass is executed that applies the simplified and unified shading to the hit surface using the dynamic lights culled by the view-space clustering pass. Even though rough reflections are not supported on transparent surfaces, rich normal map details occasionally lead to very incoherent reflection ray directions, which caused lots of incoherent memory accesses and cache pressure. A specific challenging case is shattered glass, which is fairly common in a shooting game.

46.3.3 ADDING REFLECTIONS TO RASTERIZED TRANSPARENT SURFACES

The evaluated incoming radiance toward a transparent surface from the direction of the reflection ray is not immediately applied to the surface. The actual shading of transparent surfaces happens in a separate rasterization pass. The incoming radiance is stored in a texture along with the depth of the transparent surface. When rasterizing and shading transparent objects, the depth of the shaded surface is compared to the depth value stored in the texture. When they match, the stored radiance is used instead of the environment cube map–based reflection. This approach decouples the shading of the reflection from the shading of the reflecting surface and allows using the same forward shading rasterization approach to render transparent surfaces as is used when ray tracing is disabled.

46.4 NEAR FIELD INDIRECT DIFFUSE ILLUMINATION

The implementation of ray traced indirect diffuse illumination in *Control* resembles the implementation of specular reflections in many ways. However, as using the ray traced indirect illumination also replaces modulating the precomputed global illumination with screen-space occlusion, it has two aspects. It works as ray traced ambient occlusion in addition to actually evaluating dynamic indirect diffuse lighting. The results are shown in Figure 46-10.



Figure 46-10. *Left: precomputed GI is applied on the opaque surface modulated by screen-space occlusion. Right: ray traced dynamic indirect diffuse illumination is applied on the surface in addition to the precomputed GI modulated by ray traced occlusion.*

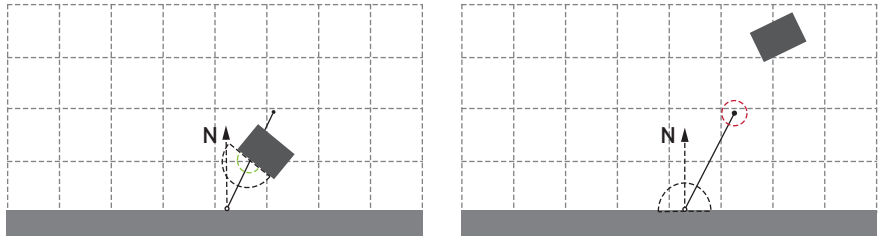


Figure 46-11. *Generated indirect diffuse illumination rays based on the G-buffer. The ray length is limited. When the ray misses, the precomputed GI is applied to the surface instead of the radiance coming from the hit surface. This makes the effect work both as ray traced ambient occlusion and as dynamic indirect diffuse illumination.*

The ray generation happens based on the rasterized G-buffer using cosine distribution for the ray directions as defined by the diffuse Lambert BRDF model. The ray length is limited to one meter. The short rays work well for resolving occlusion as they need to mostly cover only the occlusion from dynamic occluders. The occlusion caused by static large objects at larger distances is already precomputed to the GI data, which is applied to the surface when the ray misses, as illustrated in Figure 46-11. When the ray hits, radiance from the hit surface is applied instead. When evaluating the radiance, the specular BRDF is ignored in order to eliminate noise. The direction-dependent specular highlights could add a considerable amount of noise when the diffuse integral over hemisphere is approximated with only one ray per pixel.

The precomputed GI is used to approximate the second bounce for hits. Similar to the shading of specular reflections, this allows partially unified handling of hits and misses. The GI sampling code is executed regardless of whether the ray hits or misses. And similar to the specular reflections, the shading is executed in a separate compute pass using the simplified, unified material parameterization and shading model and the results of the view-space light clustering pass.

46.5 CONTACT SHADOWS

Traditional shadow maps may suffer from shadow acne due to insufficient resolution or shadow map bias. Though there are ways of mitigating these issues, ray tracing is a low-effort way to not have these problems in the first place.



Figure 46-12. *General overview of contact shadow rendering.*

Though ray traced shadows accurately solve visibility, they might not be a feasible solution for hundreds of lights due to performance reasons, especially if long rays are needed for capturing visibility in a large scene. Because shadow maps are a good and fast solution on a large scale and ray traced shadows excel with short rays, why not use both? Combining the techniques gives great image quality with high performance. Shadow maps handle most cases and ray traced shadows fills in the details. Ray tracing shadows using only very short rays makes them fast and gives the accuracy that we might be missing from shadow maps.

In *Control*, we take the following approach, shown in Figure 46-12: Regular shadow maps are rendered for all shadow casting lights. A few lights are selected for contact shadows and then (non-translucent) visibility is traced for them. The visibility buffer is denoised before it is used in lighting (discussed in Section 46.6.2). Finally, shading is done using both shadow maps and denoised contact shadows.

46.5.1 LIGHT SELECTION

In *Control*, lights are culled using a frustum volume and lighting is deferred. During the main lighting pass, the maximum intensity point lights or spotlights are recorded per pixel. However, not all pixels will be covered by any point lights or spotlights. In that case a pixel is left blank. These lights are ignored in the main lighting, but later added with visibility from both ray traced contact shadows and shadow maps.

Contact shadows are traced for lights, if any, in the maximum intensity buffer. Figure 46-13 shows an example of the maximum intensity light buffer. Black areas indicate that these regions do not have any lighting from point lights or spotlights.

46.5.2 TRACING CONTACT SHADOWS

Tracing shadows is a simple task: trace a ray from the current pixel position in world space toward a selected light. To get soft shadows, the ray direction needs to be jittered with an offset that is within the light's radius.

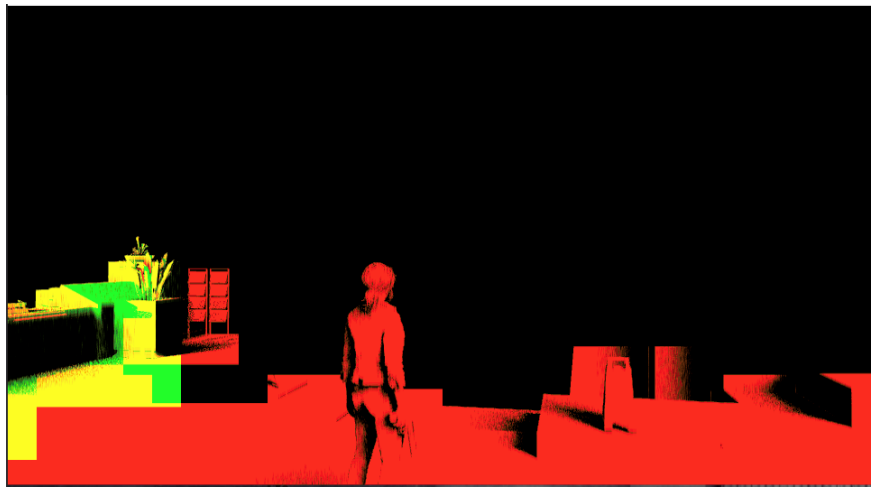


Figure 46-13. *Spotlight index is stored in the red channel and point light index in the green channel. These lights don't always cover the whole screen, and usually there are only a few different lights that dominate.*

Both spotlights and point lights are treated as spherical lights. This will create soft penumbras, but with a cost of noise, which eventually needs denoising before it can be used in lighting. Alternatively, more rays could be traced, but one ray per pixel with a well-designed denoiser and short ray distance gives good results. Denoising of contact shadows is discussed in Section 46.6.2.

The ray direction offset is calculated by first sampling a blue noise texture. This texture gives a random seed that is used to sample a concentric disk. The sample from the disk is multiplied with the light's radius and used to jitter the ray direction using the orthonormal basis of the light.

Contact shadows are only traced for opaque surfaces within the camera frustum. Hit and miss shaders return binary visibility and hit distance (`hitT`), which are both stored. Visibility from both spotlights and point lights are written to the same buffer in separate channels. Compare the results in Figure 46-14.

In *Control*, ray traced shadows are computed only for the two most significant lights, using a limited ray length and a single ray per pixel. This works very well in terms of performance and visual impact.



Figure 46-14. *Left: basic shadow map usage. Right: contact shadows add details on contacts between the floor and other objects.*

46.6 DENOISING

Denoising is an essential part of ray tracing effects. Many ray traced effects are based on Monte Carlo integration, which inherently produce noisy results. Usually, we trade performance for quality, but with a good denoiser we can have both.

Our denoising passes start by implementing a ray generation shader for each ray traced effect. For example, the style of random noise use for jittering the ray direction can affect how the resulting noisy image will look. A poorly chosen noise generator can leave a visible, recognizable pattern in the final image.

Also, the ability to discard sources of noise already while shading the rays can simplify the denoising task. For example, avoiding incoherent rays on rough surfaces and using smooth, precalculated data instead of sparsely sampled rays can result in a cleaner image, which is easier to denoise. However, not all sources of noise are avoidable and different effects may need different filters.

In *Control*, separate denoisers are implemented for reflections, indirect diffuse illumination, and contact shadows.[1] The input data for each of these effects is slightly different and can benefit from different filtering approaches. The filtering approaches used in *Control* were inspired by spatiotemporal variance-guided filtering (SVGF) [5].

46.6.1 DENOISER FOR REFLECTIONS AND INDIRECT DIFFUSE ILLUMINATION

To reduce the sources of noise in the input for reflections, we use shorter rays on rough surfaces, sample precomputed GI data on ray misses, and treat transparent reflections as mirror reflections. The ray direction is jittered with a world-space position–based random noise to diminish screen-space correlations.

Similarly, the ray generation shader for indirect diffuse illumination uses precomputed GI data to minimize noise and the aforementioned scheme for perturbing the ray direction.

The reflection and indirect diffuse illumination denoisers are the most similar and share most of the same code, but are still executed as separate passes. As a first step, they both have a firefly filter (embedded into the temporal pass), which will clamp spiky, high intensities. Firefly clamping is done by sampling the source texture intensity and clamping it with an average luminance from a lower mip level. We use different mip levels and clamping constants for reflections and indirect diffuse illumination.

After intensity clamping, regular temporal accumulation is performed with the previous frame's spatial filtering result, as shown in Listing 46-3. For the reflection denoiser, we apply variance clipping [4].

The temporal filter is followed by a spatial filter. The number of spatial filtering passes depends on the effect. For reflections, the spatial filter is executed twice per direction (horizontally and vertically) and three times for indirect diffuse illumination. The spatial filter samples the current pixel color and takes four samples with an offset (see `extendOffset` in Listing 46-4) that is extended in each pass [5, 6]. Each sample has a weight applied to it, which varies depending on which effect we denoise.

After the weight calculations, all samples are weighted and the final denoised color is resolved. On the last spatial filter iteration, we will temporally

[1] Technically, the reflection and indirect diffuse denoisers could be combined for better performance.

Listing 46-3. *Temporal filter.*

```
void TemporalFilter(...) {
    // Read source: reflections or indirect diffuse illumination.
    float3 finalColor = sourceTexture.Load(uint3(position, 0));
    // Filter high frequencies using lower mip levels of sourceTexture.
    finalColor = clampIntensity(finalColor);
    float2 previousUv = reprojectToPreviousFrame(position);
    float3 previousColor = historyColor.SampleLevel(sampler, previousUv, 0.0
        f);
    float temporalWeight = <user-defined-maximum>;
    temporalWeight *= isDepthValid(currentDepth, previousDepth);
    temporalWeight *= getVelocityWeight(currentUv, previousUv);
#if RELECTIONS
    previousColor = doVarianceClip(previousColor);
#endif
    finalColor = lerp(finalColor, previousColor, temporalWeight);
    targetTexture[position] = finalColor;
}
```

accumulate once more with the result from the first temporal accumulation pass.

After the spatial pass, we add a Fresnel component to the denoised signal—for both reflections and indirect diffuse illumination. Demodulating the Fresnel component gives us a cleaner signal to denoise.

WEIGHTING OPTIONS

Our spatial filter pass uses multiple approaches to weight samples. In addition to temporal accumulation, we use a set of filters depending on the effect. We have tuned these weights for our application. We are mostly validating or weighting input data against various surface attributes and hand-picked constants.

In the reflection denoiser, we use bilateral weights (see Section 46.6.1), filter weights [5], a weight based on hit distance, variance clipping, and a weight based on smoothness. As we are handling rough reflections mostly using precomputed GI data, which reduces input noise, we can bilaterally filter reflections with surface attributes, e.g., depth, normals, and roughness.

The indirect diffuse signal is much noisier than the reflection signal. Thus, we need to take a more relaxed approach in filtering. We mostly want to limit how far we can accept data and not strictly discard it based on surface attributes. The indirect diffuse denoiser uses bilateral weights, filter weights, and a weight based on hit info.

Listing 46-4. *Spatial filter.*

```
void SpatialFilter(...) {

    // Initialize pixel position, pixel UV, previous position, etc.
    ...
    float currentWeight = 1.0f;
    float sampleWeights[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
    float3 currentColor = sourceTexture.Load(position);

    currentWeight *= getWeightsUsingBilateralFilter(...);
    currentWeight *= getWeightsUsingFilterWeights(...);

    for (int i = 0; i < 4; ++i) {   // Samples from neighborhood
        sampleColor[i] = sourceTexture.Load(position + extentOffset(i + 1));
        // Apply a number of filters depending on
        // which effect we are denoising.
        sampleWeights[i] *= getWeightsUsingFilterA(...);
        sampleWeights[i] *= getWeightsUsingFilterB(...);
        sampleWeights[i] *= getWeightsUsingFilterC(...);
        ...
    }

    // Resolve samples.
    currentColor *= currentWeight;
    for (int i = 1; i < 4; ++i) {
        sampleColor[i] *= sampleWeights[i];
    }

    float3 finalColor = currentColor;
    for (int i = 1; i < 4; ++i) {
        finalColor += sampleColor[i];
    }

    // Normalize with total weight from this pass.
    finalColor *= inv(currentWeight + length(sampleWeights));

    // Do one more temporal pass.
    if (isLastIteration) {
        // Same logic as before
        ...
        finalColor = lerp(finalColor, previousColor, temporalWeight);
    }
    targetTexture[position] = finalColor;
}
```

We show denoised results for reflections and indirect diffuse illumination in Figures 46-15 and 46-16, respectively.

BILATERAL WEIGHTS

The bilateral weights are calculated by taking four samples with an offset from depth, normal, smoothness, and material ID buffers. Then, we calculate a weight from each sample set and finally combine these weights into one, which is returned, as shown in Listing 46-5.

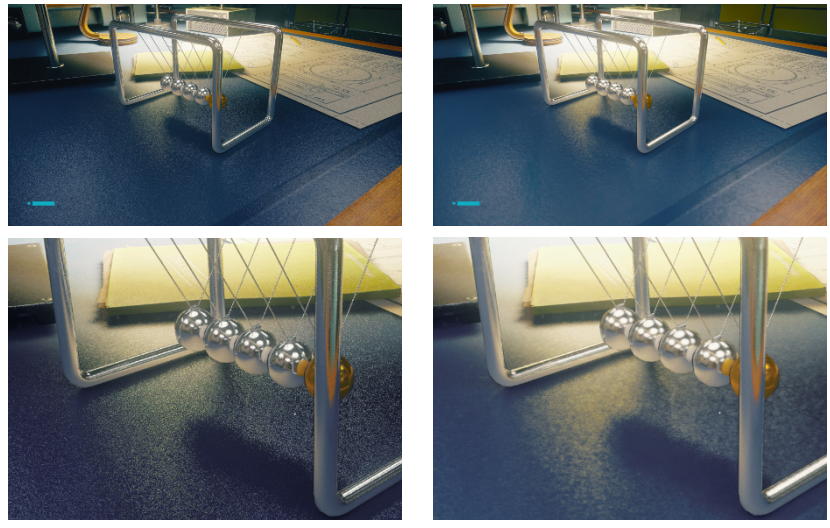


Figure 46-15. *Noisy (left) and denoised (right) reflection buffers compared.*



Figure 46-16. *Noisy (left) and denoised (right) indirect diffuse illumination buffers compared.*

46.6.2 CONTACT SHADOW DENOISER

The contact shadow ray generation shader does two things as a preparation for denoising: it uses blue noise as a random seed for the ray direction offset, and it writes out `hitT` along with the visibility. A random seed is used for sampling a concentric disk. The random sample from the disk is multiplied by the light radius and is used for offsetting the ray direction. The value `hitT` is later used in denoising.

Listing 46-5. *Bilateral weight.*

```
void getBilateralWeight(...) {
    float depth0 = depthBuffer.Load(position);
    float depth1 = depthBuffer.Load(position + offset1);
    float depth2 = depthBuffer.Load(position + offset2);
    ...
    // Repeat for other buffers.
    float4 weightDepth = abs(depth0 - float4(depth1, depth2, depth3, depth4)
        );
    ...
    // Repeat for other sample sets.
    float4 finalWeight = 1.0f;
    finalWeight *= max(<user-defined-min>, saturate(1.0f - weightDepth * <
        user-defined-multiplier>));
    ...
    // Repeat for other weights.
    return finalWeight;
}
```

The shadow denoiser (shown in Listing 46-6) is built similarly to the denoisers for reflections and indirect diffuse illumination. A temporal filter is executed first, which accumulates the previous frame's visibility data with the current frame's visibility if reprojection succeeds. The temporal filter is followed by a spatial filter, which is tailored for shadows: we know which light hit which pixel and can access the information related to that light. That information can be used to denoise shadows efficiently.

Listing 46-6. *Shadow filter.*

```
LightData centerLight; // Fill LightData struct.
...
centerLight.sigma =  GetRadiusInWorld(worldPos, centerLight.worldPosition,
     lightRadius, centerLight.hitT) * 0.6666f;

for (int i = 1; i < filterSize; i += filterStepSize) {
    samplePosition = position + int2(-i,-i) * filterDirection;
    DenoisePixel(denoisedVisibility, sumOfWeights, centerLight,
        samplePosition, currentDepth, positionInWorld);
    samplePosition = position + int2(i, i) * filterDirection;
    DenoisePixel(denoisedVisibility, sumOfWeights, centerLight,
        samplePosition, currentDepth, positionInWorld);
}

denoisedVisibility /= sumOfWeights;
denoisedVisibility = saturate(denoisedVisibility);

// Write out denoisedVisibility.
```

The spatial filter used in *Control* was heavily inspired by the Gameworks spatial shadow filter by Story and Liu [2]. The filter is separable, executed

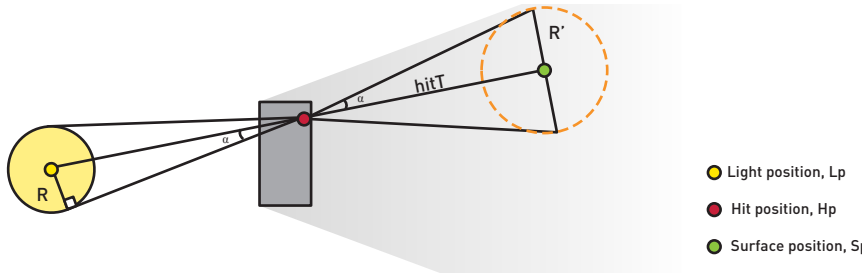


Figure 46-17. *The light radius can be projected onto the shadowed surface using* `hitT`.

once per direction. The idea of the spatial filter is to check per pixel which light potentially created the shadow, approximately project the light radius on the shadowed surface, and use that information to weigh neighboring samples. For sample weighting we used Gaussian weights with sigma calculated from the light radius in world space [3].

First, a filter kernel is initialized retrieving light data for the current pixel, which we refer to as the *center*. We prepare the kernel with the light position in world space, radius, index, visibility, `hitT`, and Gaussian deviation:

```
struct LightData {
    float3  lightPositionInWorld;
    float   hitT;
    float   lightRadius;
    float   sigma;
    float   visibility;
    uint    lightIndex;
};
```

The Gaussian deviation is calculated from the projected light radius (see Figure 46-17). The projected light radius can be calculated using the surface position, light position, and `hitT`. Please refer to Listing 46-7.

Listing 46-7. *Projected light radius.*

```
float GetRadiusInWorld(float3 surfacePosition, float3 lightPosition, float
     lightRadius, float hitT) {
    float3 surfaceToLight = lightPosition - surfacePosition;
    float3 hitPosition = surfacePosition + hitT * normalize(surfaceToLight);
    float hitDistance = length(lightPosition - hitPosition);
    float lightHalfFov = asin(lightRadius / hitDistance);
    return tan(lightHalfFov) * hitT;
}
```

After we've initialized our filter kernel, we can start sampling. We use a kernel radius of eight pixels and take two pixel-wide steps. On each step we call `DenoisePixel`, which samples visibility at the pixel and validates it against the depth and light indices. (See Listing 46-8.) If the visibility data is from the same light as our kernel center, we can potentially use it.

Listing 46-8. *DenoisePixel.*

```
void DenoisePixel(inout float denoisedVisibility, inout float sumOfWeights,
     LightData centerLight, LightData centerPoint, uint2 sampePosition,
     float centerDepth, float3 positionInWorld) {

    // Check point light and spotlight indices at sample pixel.
    uint sampleLightIndex = GetLightIndex(sampePosition);
    float sampleDepth = GetDepth(sampePosition);

    // Sample raw contact shadow buffer.
    float sampleVisibility = GetVisibility(sampePosition);

    // Check sample validity in depth.
    uint isValid = IsValid(sampleDepth, centerDepth) ? 1 : 0;

    float sampleWeight = 0.0f;
    float distanceSampleToCenter = length(positionInWorld -
        samplePositionInWorld);

    // We can use this sample if it's visibility data is
    // from the same light as the center.
    if (sampleLightIndex == centerLight.index && isValid) {
        // Calculate Gaussian weight in world space.
        sampleWeight = GetWeightInWorld(distanceSampleToCenter, centerLight)
            ;
    }

    denoisedVisibility += sampleVisibility * sampleWeight;
    sumOfWeights += sampleWeight;
}
```

We can only use a sample's information if it originates from the same light as the center pixel's sample. In theory, when the maximum intensity light buffer is created, each pixel could get contributions from a different light if there were numerous lights in the scene. In this case, we would not be able to denoise, because all visibility information in each pixel would originate from different lights. Luckily, that was not a common lighting setup for *Control*.

For each valid sample we calculate the sample distance from the kernel center in world space and use it to calculate a Gaussian weight:

```
float GetWeightInWorld(float length, float sigma) {
    if (sigma == 0.0f)
        return (length == 0.0f) ? 1.0f : 0.0f;
    return exp(-(length * length) * rcp(2.0f * sigma * sigma));
}
```

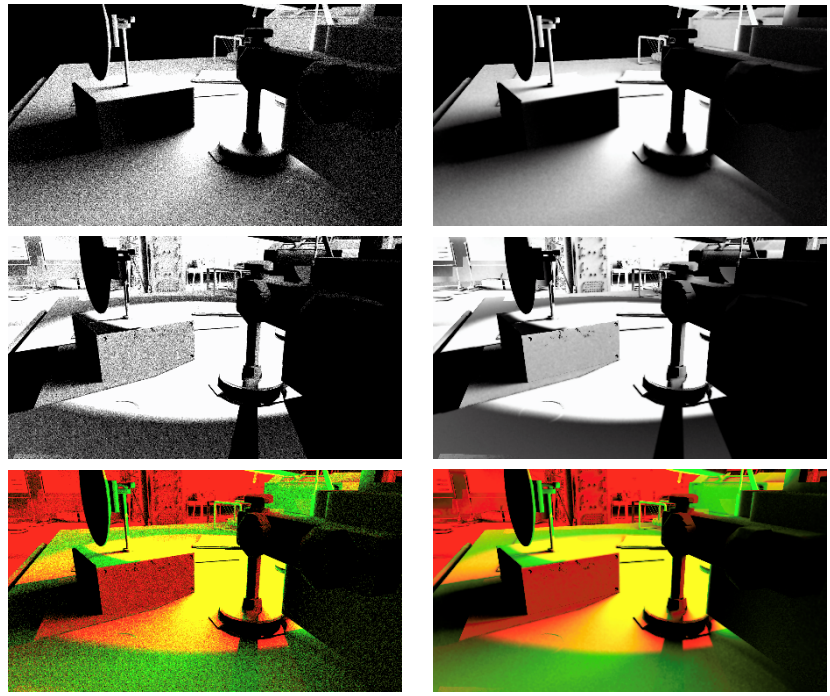


Figure 46-18. *Left: noisy. Right: denoised. Top: spotlight shadow. Middle: shadows from a few point lights. Bottom: combined shadow buffers.*

When all filter steps are done, we write out the denoised visibility and use it in the lighting. Compare the results shown in Figure 46-18.

46.7 PERFORMANCE

Ray tracing performance varies from frame to frame depending on how much of the screen is covered by transparent surfaces and how much of the screen is covered by lights selected for contact shadows. We captured two representative frames at a resolution of 2560×1440 pixels on NVIDIA RTX 3090 to give an example of ray tracing performance. In our example cases, shown in the Figure 46-19, all ray tracing effects take around 8.9 ms of the 19 ms total frame time and 7.1 ms of the 16 ms total. This time is divided into tracing rays, shading the hit results, and denoising. That is a significant fraction of the frame time, but bear in mind that tracing rays enables effects that would not be otherwise possible and adds a lot to the realism of the scene.



Figure 46-19. *Two frames captured at a resolution of* 2560 × 1440 *pixels on NVIDIA RTX 3090 for performance measurements.*

Table 46-1 shows the timing for the individual passes of the ray tracing effects. As we can see, denoising usually takes almost as long as if not even longer than tracing rays. Tracing performance is largely dependent on the ray count and the geometric content of the frame, and denoising only depends on the rendering resolution. Spending an equal amount of time in denoising as in ray tracing might sound a lot, but in practice it offers a good balance between performance and image quality. If we increased the ray count, performance would be impacted significantly. Going from one to two rays per pixel often doubles the time spent in tracing, but likely only has a modest impact on image quality. We argue that a fairly low ray count combined with domain-specific denoisers is the current sweet spot between image quality and performance.

	Frame 1	**Frame 2**
Pass	**Time (ms)**	**Time (ms)**
Acceleration structure building (async.)	0.6	1.0
Reflection ray tracing	1.0	1.4
Reflection shading	1.4	1.1
Reflection denoising	0.8	0.8
Transparent reflection ray tracing	0.8	0.3
Transparent reflection shading	0.7	0.1
Indirect diffuse ray tracing	0.8	0.7
Indirect diffuse shading	0.8	0.6
Indirect diffuse denoising	1.1	1.0
Contact shadow ray tracing	0.8	0.4
Contact shadow denoising	0.7	0.7
Total Cost:	8.9	7.1

Table 46-1. *Frame time spent on different ray tracing effects at a resolution of* 2560 × 1440 *pixels on NVIDIA RTX 3090. The acceleration structure build time is not included in the total cost because it performed asynchronously.*

46.8 CONCLUSIONS

In this chapter, we have shown how ray traced reflections, near field indirect diffuse illumination, and contact shadows can be implemented in a hybrid renderer. By carefully tuning the input signal and designing domain-specific denoisers, we have succeeded in adapting these effect to a visual quality and performance level suitable for a shipped game title. We are very happy with the first launch of the game *Control*. We show that ray tracing can be used to enhance an existing rendering pipeline that is deployed also on platforms without hardware-accelerated ray tracing. Without too much hassle, we were able to bring out visual details and accuracy not possible with traditional rasterization techniques. The effects and their implementation fit comfortably to the game. We look forward to how ray tracing can be utilized in future projects.

REFERENCES

[1] Aalto, T. The latest graphics technology in Remedy's Northlight engine. Presentation at Game Developers Conference, https://www.dropbox.com/s/ni32kzk0twjzwo0/RemedyRenderingGDC18.pptx?dl=0, 2018.

[2] Gruen, H., Roza, M., and Story, J. "Shadows" of the Tomb Raider: Ray tracing deep dive. Presentation at Game Developers Conference, https://www.gdcvault.com/play/1026163/-Shadows-of-the-Tomb, 2019.

[3] Mehta, S., Wang, B., and Ramamoorthi, R. Axis-aligned filtering for interactive sampled soft shadows. *ACM Transactions on Graphics*, 31(6):163:1–163:10, 2012. DOI: 10.1145/2366145.2366182.

[4] Salvi, M. An excursion in temporal supersampling. Presentation at Game Developers Conference, 2016.

[5] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, 2:1–2:12, 2017.

[6] Smal, N. and Aizenshtein, M. Real-time global illumination with photon mapping. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 24, pages 409–436. Apress, 2019.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 47

LIGHT SAMPLING IN QUAKE 2 USING SUBSET IMPORTANCE SAMPLING

Tobias Zirr
Karlsruhe Institute of Technology

ABSTRACT

In the context of path tracing, to compute high-quality lighting fast, good stochastic light sampling is just as important as light culling is in the context of raster graphics. In order to enable path tracing in *Quake 2*, multiple solutions were evaluated. Here, we describe the light sampling solution that ended up in the public release. We also discuss its relation to more recent approaches like ReSTIR [4], real-time path guiding [11], and stochastic lightcuts [19]. Finally, we leverage the power of variance reduction techniques known from offline rendering, by providing an extension of our stochastic light sampling technique that allows use of *multiple importance sampling* (MIS). The resulting algorithm can be seen as a variant of stochastic MIS, which was recently proposed in the framework of continuous MIS [34]. To this end, we derive additional theory to introduce pseudo-marginal MIS, allowing for effective variance reduction by marginalization with respect to only parts of the sampling process.

47.1 INTRODUCTION

The Q2VKPT project—leveraging ray tracing to bring unified solutions for the simulation and filtering of all types of light transport into a playable game—was created by Christoph Schied [25] to build an understanding of what is already feasible, and what remains to be done for future ray traced game graphics. The release of GPUs with ray tracing capabilities has opened up new possibilities, yet making good use of ray tracing remains nontrivial: ray tracing alone does not automatically produce realistic images. Light transport algorithms like path tracing can be used for that, realistically simulating the complex ways that light travels and scatters in virtual scenes. However, though elegant and powerful, naive path tracing is also very costly and takes a long time to produce stable images. Even when using smart

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_47

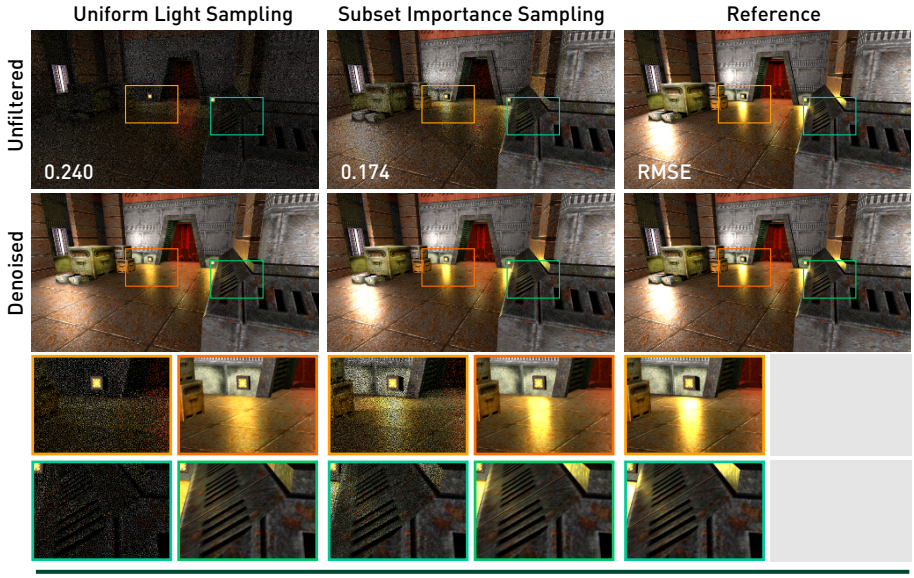


Figure 47-1. *Path tracing in Q2VKPT at one sample per pixel, with and without denoising (using Adaptive Spatiotemporal Variance-Guided Filtering, ASVGF [27]). Sampling all lights in the scene uniformly leads to poor shading and shadow quality, both with and without denoising (left). For the public release of Q2VKPT, we implemented stratified resampled importance sampling (RIS) [30, 5], performing approximate product importance sampling of light and material contributions within stochastic subsets of all lights (middle).*

adaptive filters [26, 7, 27] that reuse as much information as possible across many frames and pixels—similarly to *temporal antialiasing* (TAA)—care has to be taken to keep variance manageable in order to efficiently produce clean and stable images.

Monte Carlo (MC) techniques like path tracing work by tracing random light paths that connect the camera and light sources via scattering surfaces in the scene. Ray tracing is used to resolve visibility: for each scattering interaction, it finds the next visible surface in a given direction. For the resulting shading estimates to be robust (i.e., for them to efficiently converge to the correct result with few path samples), the random paths need to represent the actual illumination in the scene well. However, such good *importance sampling* (IS), ensuring that the path density closely matches the distribution of light transport in the scene, is hard: perfect importance sampling would require predicting the distribution of (indirect) light via arbitrary scattering interactions in advance.

In Q2VKPT, following common practice in (offline) movie rendering [13, 6, 14, 9], path tracing is used with additional light sampling (*next event*

estimation (NEE)) in order to reliably find the relevant lights for each surface in a scene: At each visible surface point, two rays are traced. One is a shadow ray traced toward a random point on a random light source. This point on the emitter is sampled in a way that makes it representative for the entire illumination in the scene (i.e., the probability of sampling specific emitters is proportional to their shading contribution; this is crucial and the concern of this chapter). To recursively accumulate multi-bounce illumination, another ray points in a random direction sampled proportionally to the scattering of the shaded material. The next visible surface in this direction is then shaded in the same way.

Similarly to light culling in real-time rendering, picking the right lights for each surface point is crucial for image quality: in the MC path tracing framework, picking the wrong lights too often results in highly unreliable shading estimates, which then force reconstruction filters to suppress the resulting outliers, removing all the details that the path tracer was supposed to produce in the first place.

47.2 OVERVIEW

In this chapter, we present the constant-time light sampling algorithm implemented in Q2VKPT. The algorithm uses stratified sampling to obtain random subsets with constant size for every frame and pixel, sampled from a potentially long list of relevant light sources. Thus, it quickly hits all light sources over time, while the stratified sampling, with the right locality in light lists, keeps subsets representative of the full illumination in the scene. Within a stochastic subset, importance sampling according to the precisely estimated influence of each light source can be done in a controlled time budget.

In contrast to more recent approaches like ReSTIR [4], our approach is more primitive and thus less optimal, but independent of spatiotemporal data structures (see also Section 47.3)—how to best adapt ReSTIR to multiple bounces in this regard is a question for future research. During development, Q2VKPT was also tested with advanced hierarchical light sampling approaches [10, 20, 19] inspired by movie production: by clustering lights hierarchically, the influence of many lights can be estimated at once, allowing for quick exclusion of far away, dim lights and of lights pointing the wrong way. However, in our experience, such estimates are hard to get precise (see also Section 47.3).

For this chapter, an additional challenge we address is the implementation of *multiple importance sampling* (MIS) with our subset sampling technique. MIS is crucial to limit variance in some common cases, e.g., for specular highlights of nearby reflected emitters and for emitters in close proximity to lit surfaces. Without MIS, these cases require aggressive clamping, potentially hampering the physically based appearance that is expected from a path tracing–based renderer. To integrate MIS with our technique, we build on the framework of stochastic MIS (recently introduced in the context of *continuous MIS*), and we propose pseudo-marginal MIS to allow effective variance reduction using only the readily available information of a stochastic subset of lights, i.e., without requiring expensive full marginalization of probability densities.

47.3 BACKGROUND

Scenes in games and movie productions can contain large amounts of light sources, and as such, sampling of light sources has been a long-standing topic of research both in academia and production rendering. Our goal is to sum the contribution of all light sources $\mathcal{L}_l$ for $l = 0, \ldots, K-1$, emitting light $L_l(\mathbf{x}, \mathbf{y})$ from points $\mathbf{y} \in \mathcal{L}_l$ to *each* shading point $\mathbf{x}$,[1] given its normal $\mathbf{n}$, its material as defined by the *bidirectional scattering distribution function* (BSDF) f_r, and a view direction $\mathbf{o}$:

$$L_o := \sum_{l=0}^{K-1} \int_{\mathcal{L}_l} f_r\left(\mathbf{o}, \mathbf{x}, \mathbf{i}\right)\ \mathbf{n}^T\mathbf{i}\ V(\mathbf{x}, \mathbf{y})\ L_l(\mathbf{x}, \mathbf{y})\, \mathrm{d}\mathbf{y}, \quad \text{where } \mathbf{i} = \frac{\mathbf{y} - \mathbf{x}}{\|\mathbf{y} - \mathbf{x}\|}, \tag{47.1}$$

and the visibility $V(\mathbf{x}, \mathbf{y})$ of $\mathbf{y}$ from $\mathbf{x}$ is determined via ray tracing to obtain accurate shadowing. Computing accurate shading for all shading points and a large number of light sources would be prohibitively expensive. Instead, Monte Carlo integration evaluates a random variable F that computes the shading integrand for only one random point on only one randomly chosen light source for each shading point, in a way such that the expected value of F happens to coincide with the accurate shading by all light sources:

$$F := \frac{f_r\left(\mathbf{o}, \mathbf{x}, \mathbf{i}\right)\ \mathbf{n}^T\mathbf{i}\ V(\mathbf{x}, \mathbf{y})\ L_l(\mathbf{x}, \mathbf{y})}{p(\mathbf{y}|\mathcal{L}_l)P(\mathcal{L}_l)}, \quad \mathbf{E}_{p(\mathbf{y}|\mathcal{L}_l)P(\mathcal{L}_l)}[F] = L_o. \tag{47.2}$$

As long as the chance of sampling is nonzero for all the points on the light sources that contribute nonzero shading to the point $\mathbf{x}$, we are free to choose

[1] For an area light, $L_l(\mathbf{x}, \mathbf{y}) = (|\mathbf{n}_l^T\mathbf{i}|\ L_l(\mathbf{y}, -\mathbf{i}))/\|\mathbf{y} - \mathbf{x}\|^2$, where $\mathbf{n}_l$ is the normal of the emitting surface and $L_l(\mathbf{y}, -\mathbf{i})$ is its radiance at $\mathbf{y}$ toward $\mathbf{x}$. For a point light, $L_l(\mathbf{x}, \mathbf{y}) = (\delta(\mathbf{y} - \mathbf{y}_l)\ I_l(-\mathbf{i}))/\|\mathbf{y} - \mathbf{x}\|^2$, where $\mathbf{y}_l$ is its position and $I_l(-\mathbf{i})$ is its intensity toward $\mathbf{x}$. For a directional light, $L_l(\mathbf{x}, \mathbf{y}) = \delta(\mathbf{y} - \mathbf{y}_l(\mathbf{x}))\ E_l(\mathbf{x})$, where $E_l(\mathbf{x})$ is its irradiance toward $\mathbf{x}$ and $\mathbf{y}_l(\mathbf{x})$ projects $\mathbf{x}$ onto its orthogonal plane.

any sampling strategy: In F, divide by the probability $P(\mathcal{L}_l)$ of randomly choosing the light source $\mathcal{L}_l$, and divide by the probability density $p(\mathbf{y}|\mathcal{L}_l)$ of randomly choosing the point $\mathbf{y}$ on $\mathcal{L}_l$, then compensate for how often each of the samples occurs. Both divisions are canceled out in the expected value.

In order to obtain good shading estimates fast, we want to limit the potential deviation of F from the accurate shading L_o (typically quantified by the variance $\mathbf{V}[F]$). This is achieved when the light source $\mathcal{L}_l$ and $\mathbf{y} \in \mathcal{L}_l$ are sampled often where their contribution to the shading point $\mathbf{x}$ is high: ideally, we want the probability density $p(\mathbf{y}|\mathcal{L}_l)P(\mathcal{L}_l)$ of light sampling to cancel out the shading integrand [29], such that F always equals the correct result for *any* random samples $\mathcal{L}_l, \mathbf{y} \in \mathcal{L}_l$.

47.3.1 LIGHT RESAMPLING

To achieve such *importance sampling* in real-time path tracing, Bikker [2] uses *resampled importance sampling* (RIS) [30, 5], where first a set of candidate points on light sources is sampled, then their contribution is estimated to sample one final point, using a distribution function proportional to their contribution relative to the other candidate points. For direct illumination, Bitterli et al. [4] propose the ReSTIR algorithm, which optimizes and refines this approach in many ways: Based on the framework of reservoir sampling [8], they avoid storing any candidate points, resampling candidates such that at any time only one tentative final point, surviving the sampling process, needs to be stored. Furthermore, they observe that resampling results can be reused across pixels and even multiple frames, applying resampling hierarchically. By this parallel distribution of sampling efforts over space and time, ReSTIR quickly achieves importance sampling with respect to very large numbers of light sources and candidate emitter points, leading to high-quality, low-variance shading results with little computational overhead.

In the public release of Q2VKPT, our stochastic light subset sampling without MIS bears similarities to Bikker's approach, but adds a stratified sampling scheme that leads to better screen-space error distribution, uses fewer random variables, and avoids creating many candidate points on light sources (see Section 47.4.3). Internally, we also tested reservoir sampling to avoid explicit storage of candidate contributions (see Section 47.4.4), but because it reduced the benefits of using blue noise pseudo-random numbers, we decided against it in the public release. As we show in this chapter, our approach can be extended to allow MIS [31], which efficiently reduces

variance and thus potential energy loss due to clamping, especially for reflective materials. In comparison to ReSTIR, our algorithm is more primitive and less optimal, but in contrast, our approach also handles indirect illumination easily, requiring no changes.

47.3.2 HIERARCHICAL LIGHT SAMPLING

Shirley et al. [29] describe sampling strategies for different light source shapes. To reduce the amount of light sources they need to consider, they precompute importance stored in an octree. It has since become a common approach in offline rendering to organize light sources into a tree data structure, which is then typically stochastically traversed [13, 18] for efficient importance sampling among many lights. Depending on performance trade-offs, lights are clustered hierarchically by either higher-quality *bounding volume hierarchies* (BVHs), mixed-quality two-level BVHs [21], or faster-to-construct *linear-time BVHs* (LBVHs) [19].

A difficulty of such hierarchical importance sampling schemes is estimating the contribution of many lights from purely aggregate information in coarser levels during traversal: when a shading point is close to a cluster of lights, or even contained therein, the contribution of each contained light may depend strongly on the location and orientation of the shading point. To this end, Estevez and Kulla [10] complement contribution estimation heuristics with reliability heuristics during traversal, collecting lights from all subtrees whenever an aggregate contribution estimate is deemed unreliable. Similarly, stochastic light cuts [19] combine the ideas of light cuts [33] (representing the emissions of many lights approximately by fewer aggregate lights, clustering hierarchically and refining adaptively during rendering) and of light sampling hierarchies (replacing any selected aggregate lights in the light cut by hierarchically sampling leaf lights to recover unbiasedness).

Variations of these hierarchical algorithms vary in their aggregate reliability heuristics, the number of lights selected, and the data structures used. We refer to Moreau and Clarberg [20] and Estevez and Kulla [10] for a broader overview. As common in real-time rendering, to further improve sampling and performance at the cost of introducing bias, the number of light sources considered at each point can be reduced by restricting the emission range [28, 3, 14].

In Q2VKPT, we experimented with light hierarchies following Moreau and Clarberg [20], adapting the heuristics of Estevez and Kulla [10] to real-time

rendering. However, lacking the adaptive splitting heuristic [10] (which would require costly traversal of all subtrees when encountering unreliable contribution estimates), the algorithm was hard to get robust. We often saw aggregate contribution heuristics failing and leading to visible noise artifacts in the resulting image. With the right tuning, a two-phase parallel approach of first splitting then stochastic traversal as in stochastic light cuts [19] might work for certain GPU applications. However, in either form, stochastic tree traversals cause incoherent memory access patterns. Finally, in animations, tree updates can cause drastic changes of split quality metrics and thus tree topology, making the sampling quality temporally inconsistent. Our simpler light subset selection algorithm avoids these caveats.

47.3.3 SAMPLE REUSE AND GUIDING

Similarly to how ReSTIR [4] distributes importance sampling over multiple pixels and frames, caching and reusing information for improved importance sampling has been done in more general settings, e.g., as importance caching [15] of light importance distributions on a sparse set of surface points, distributed in the scene before rendering. Caching has found its way into production for both direct [6] and indirect [32, 22] illumination. Dittebrandt et al. [11] come up with data structures and compression schemes to support online learning of light sampling distributions in real time, which also works for indirect bounces. Their paper shows results for *Quake 2* RTX. Typical challenges of guiding, learning, caching, and reusing algorithms are the detection of similarity, applicability, and expiration of cached or shared information, which we avoid by the simple approaches in this chapter.

47.4 STOCHASTIC LIGHT SUBSET SAMPLING

Because tracing rays for shadow tests is still a costly operation, we follow common path tracing practice and select only one light source per shading point. It may seem like we can skip shading computations for many light sources altogether, however this would not lead to high-quality shading but to a lot of noise: We still need to approximate shading contributions of all lights that we choose from. Only then, we can choose each light with the right probability proportional to its contribution [29], ensuring MC sample values that are close to the correct result.

The challenge we address in this section is to reduce the number of light sources for which we approximate shading contributions, while keeping the

probability of sampling each light approximately proportional to its contribution relative to all light sources. In Q2VKPT, for each shading point we have a (possibly long) *light list* of emitters that are potentially relevant for shading that point. We build these lists from the *potentially visible set* (PVS) for each cluster of level geometry (binary space partitioning (BSP) node) at load time. Nearby dynamic lights are appended at runtime. Building lists of relevant lights for level geometry is a known problem to game developers with various solutions (parallel grid/cluster culling based on proximity heuristics, potentially visible sets, etc.).

To achieve our goal of cutting down on the long lists of relevant light indices J at a shading point **x**, let us split the contribution of all K relevant light sources into S parts, iterating through K/S light indices J_s in each part:

$$L_o = \sum_{s=0}^{S-1} \sum_{l \in J_s} \int_{\mathcal{L}_l} f_r\left(\mathbf{o}, \mathbf{x}, \mathbf{i}\right) \; \mathbf{n}^T\mathbf{i} \; V(\mathbf{x}, \mathbf{y}) \; L_l(\mathbf{x}, \mathbf{y}) \, \mathrm{d}\mathbf{y}. \tag{47.3}$$

This allows us to construct a corresponding Monte Carlo estimator F_s that computes L_o by first randomly selecting a subset of lights J_s and then sampling a light index l from the subset only:

$$F_S := \frac{f_r\left(\mathbf{o}, \mathbf{x}, \mathbf{i}\right) \; \mathbf{n}^T\mathbf{i} \; V(\mathbf{x}, \mathbf{y}) \; L_l(\mathbf{x}, \mathbf{y})}{p(\mathbf{y}|\mathcal{L}_l)P(l|J_s)P(s)}, \quad \mathbf{E}_{p(\mathbf{y}|\mathcal{L}_l)P(l|J_s)P(s)}[F_S] = L_o. \tag{47.4}$$

As we want to avoid computations for more than one subset J_s, we choose one s uniformly at random ($P(s) = 1/S$). In order to still obtain high-quality, low-variance results, each subset J_s should be representative of all relevant light indices J; that is, we would like to achieve approximately the same probabilities sampling from J_s as sampling from the full distribution:

$$P(l|J_s)P(s) \approx P(l|J), \quad \text{where } J = \bigcup_{s=0}^{S-1} J_s \text{ and } J_s \cap J_{s'} = \emptyset. \tag{47.5}$$

Within the shorter lists J_s, it becomes feasible to perform product importance sampling for sampling $l \in J_s$, i.e., to sample each contained light proportional to its predicted shading contribution (neglecting visibility). If the subset J_s is indeed representative of J, the procedure will not produce much more noise than sampling from the full light list J.

47.4.1 PRACTICAL STRIDED SUBSETS

In Q2VKPT, we use constant-size subsets of (maximum) length R, splitting the list J of K relevant light indices for a shading point into $S = \lceil K/R \rceil$ subsets J_s. If

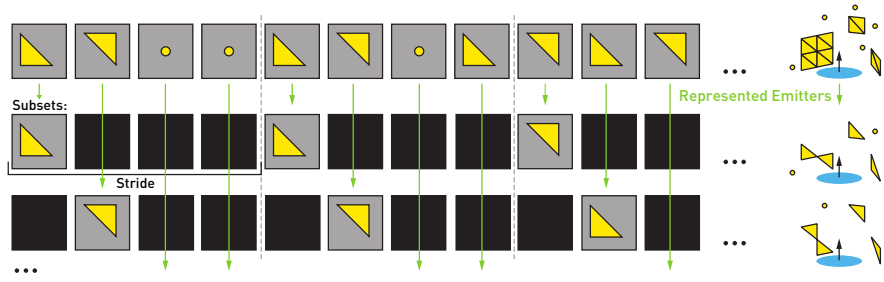


Figure 47-2. *Our light sampling performs approximate product importance sampling by importance sampling from strided "representative" subsets of the full light list for each shading point.*

the full list J of lights is small, there will be only one subset and the light sampling is optimal. For subsets of long lists J to stay representative, we choose a *strided* subset selection strategy as illustrated in Figure 47-2; we pick every Sth light from J with varying initial offsets s:

$$J_s := J\left[s + iS | i \in \mathbb{N}_0\right]. \tag{47.6}$$

The reasoning behind this strategy is based on the way our light lists are constructed in Q2VKPT: Due to our light geometry being sourced from the game level's hierarchical space partitions, spatially colocated emitters are often colocated in memory. Thus, a strided subset of light sources often still gives a good idea of their spatial distribution in the scene.

An important advantage of choosing the subset by one offset s is that we can benefit from low-discrepancy random variables: We use the same random variable to first select one subset out of S subsets, then rescale the interval that maps to offset s such that the random values contained by it can be reused to sample the light within the selected subset. If we map exactly one random variable interval to every light source (proportional to its contribution), then the resulting noise will show the pleasant dither pattern of low-discrepancy points (they are optimized to sample intervals well).

47.4.2 WORST-CASE VARIANCE ANALYSIS

The values of F_S can be bounded relative to the values of an idealized estimator F^*, which would perform importance sampling with respect to all relevant lights, sampling with the probability distribution function (PDF) $p(Y|\mathcal{L}_l)P(l|J)$:

$$F^* = \frac{f_r\left(\mathbf{o}, \mathbf{x}, \mathbf{i}\right)\ \mathbf{n}^T\mathbf{i}\ V(\mathbf{x}, \mathbf{y})\ L_l(\mathbf{x}, \mathbf{y})}{p(\mathbf{y}|\mathcal{L}_l)P(l|J)}, \tag{47.7}$$

$$F_S = F^*\frac{P(l|J)}{P(l|J_s)P(s)} = F^* S\frac{\sum_{i\in J_s} C(\mathcal{L}_i)}{\sum_{i\in J} C(\mathcal{L}_i)} \leq \lceil K/R\rceil F^*, \tag{47.8}$$

for estimated contributions $C(\mathcal{L}_l)$ of lights $\mathcal{L}_l$ to the shading point, such that $P(l|J) = C(\mathcal{L}_l)/\sum_{i\in J} C(\mathcal{L}_i)$. It follows from $P(s) = 1/S$ in our sampling procedure that, in the worst case, our subset sampling causes at most $S = \lceil K/R\rceil$ times higher error than ideal importance sampling, while guaranteeing constant runtime. In Q2VKPT, we choose $R = 8$ representatives and thus our subset count S is small. In practice, the variance is even lower: when each subset is representative of the full illumination, it is close to optimal (then $F_S \approxeq F^*$).

47.4.3 TWO-SWEEP ALGORITHM

Listing 47-1 provides pseudocode for the direct implementation of our light sampling algorithm, illustrated in Figure 47-2. Our light lists are concatenated in one long array `light_pointers`. Given a shading point **x**, the enclosing potentially visible set cluster provides an index range [`light_list_begin`, `light_list_end`) for the relevant lights. The contribution of contained light sources is predicted by the function `light_contrib`, as detailed in Section 47.4.5.

First, we split the potentially long list of K relevant lights into S representative parts of length R = `MAX_SUBSET_LENGTH`. We do this using the number of parts as a stride, such that all `subset_stride`-th lights in the list become part of the same subset. We then randomly select one of these subsets by sampling a start offset `subset_offset` relative to the beginning of the light list.

In a first loop over all lights in the selected (strided) subset, we compute a conservative estimate for the contribution of each subset light, which we store in a fixed-length importance sampling array `is_weights` and sum up to obtain a normalization constant for random sampling. Afterward, we determine a random threshold for sampling one of the subset lights with a probability proportional to its contribution weight. We do this in a second loop, reiterating the stored contribution weights until the prefix weight mass reaches the random threshold.

Finally, we compute the probability of sampling the chosen light. This probability is a coefficient that can afterward be multiplied to the PDF of sampling one point on the chosen emitter, to obtain the final light sampling PDF.

Listing 47-1. *Sampling a light source from a strided stochastic subset in two sweeps, using one random variable ξ_1.*

```
S = ⌈(light_list_end-light_list_begin) / MAX_SUBSET_LEN⌉
subset_stride = S
subset_offset = ⌊ξ1 * S⌋
ξ1 = ξ1 * S - ⌊ξ1 * S⌋

total_weights = 0
float is_weights[MAX_SUBSET_LEN]

light_idx = light_list_begin + subset_offset
for (i = 0; i < MAX_SUBSET_LEN; ++i) {
    if (light_idx >= light_list_end) {
        break
    }
    w = light_contrib(v, p, n, light_pointers[light_idx])
    is_weights[i] = w
    total_weights += w

    light_idx += subset_stride
}

ξ1 *= total_weights
mass = 0

light_idx = light_list_begin + subset_offset
for (i = 0; i < MAX_SUBSET_LEN; ++i) {
    if (light_idx >= light_list_end) {
        break
    }
    mass = is_weights[i]

    ξ1 -= mass
    if not (ξ1 > 0) {
        break
    }
    light_idx += subset_stride
}

probability = mass / (total_weights * S)
return (light_pointers[light_idx], probability)
```

47.4.4 ONE-SWEEP ALGORITHM

The requirement of intermediately storing contribution weights can increase register and/or memory pressure, depending on the compiler and architecture. We can avoid this storage and iterating over the light subset twice (at the cost of potentially less stratified sampling, e.g., if blue noise is used for random variables).

Listing 47-2. *Sampling a light source from a strided stochastic subset in one sweep, using one random variable ξ_1.*

```
1  S = ⌈(light_list_end-light_list_begin) / MAX_SUBSET_LEN⌉
2  subset_stride = S
3  subset_offset = ⌊ξ1 * S⌋
4  ξ1 = ξ1 * S - ⌊ξ1 * S⌋
5
6  selected = (light_idx: -1, mass: 0)
7  total_weights = 0
8
9  light_idx = light_list_begin + subset_offset
10 for (i = 0; i < MAX_SUBSET_LEN; ++i) {
11     if (light_idx >= light_list_end) {
12         break
13     }
14     w = light_contrib(v, p, n, light_pointers[light_idx])
15     if (w > 0) {
16         τ = total_weights / (total_weights+w)
17         total_weights += w
18
19         if (ξ1 < τ) {
20             ξ1 /= τ
21         } else {
22             selected = (light_pointers[light_idx], w)
23             ξ1 = (ξ1-τ) / (1-τ)
24         }
25         ξ1 = clamp(ξ1, 0, MAX_BELOW_ONE) // Avoid numerical problems.
26     }
27
28     light_idx += subset_stride
29 }
30
31 probability = selected.mass / (total_weights * S)
32 return (selected.light, probability)
```

For this, we use an incremental sampling scheme (reservoir sampling [8]) that retains the same probabilities of sampling each light, but redistributes the value range of the random variable ξ_1 differently: for each iterated light, a decision is made whether to keep the previously selected light or to switch to the current light as the next candidate for the final sampled emitter.

Listing 47-2 provides the altered pseudocode for the one-sweep, incremental sampling implementation of our light sampling algorithm. The threshold τ computed in line 16 ensures that the final probability of keeping the light source stored in `selected` as the final sampled emitter is still exactly proportional to its contribution weight: given the probability $P_s(L_i) = 1 - \tau$ of selecting emitter L_i in step i, we find the final probability $P(L_i)$ of selecting and

keeping L_i to be

$$P_s(L_i) = 1 - \tau = \frac{w_i}{\sum_{j=0}^{i} w_j} \tag{47.9}$$

$$\Rightarrow P(L_i) = P_s(L_i) \prod_{j=i+1} \left(1 - P_s(L_j)\right) = \frac{w_i}{\sum_{j=0}^{\#J_s-1} w_j}. \tag{47.10}$$

Besides this, Listing 47-2 closely follows what we saw in Listing 47-1.

47.4.5 PREDICTING THE CONTRIBUTION OF LIGHT SOURCES

Making a good prediction for the contribution of a single light source $\mathcal{L}_l$ to a given shading point $\mathbf{x}$ (i.e., evaluating Equation 47.1 with only one fixed l) can be a challenge in itself, depending on the involved materials, the visibility, and the shape of $\mathcal{L}_l$. As it is unpredictable, we neglect visibility (note that more recent approaches like ReSTIR [4] and real-time path guiding [11] improve on this). For predicting the unshadowed contribution, we use a rather simplistic heuristic: First, we evaluate the BRDF of the shading point for the center of the light source, clipping the resulting light direction to the horizon of the shaded surface. In order to prevent misguiding by the peaks of highly reflective materials, we limit the material roughness in this context. All our lights are triangles. To account for their extent, we compute the solid angle covered by the light source, projecting its triangle onto the sphere around $\mathbf{x}$. A precise formula is given by Arvo [1]. To conservatively bound the cosine, we compute the dot product of the normal $\mathbf{n}$ at $\mathbf{x}$ and the direction toward the highest light vertex above the shading horizon.

47.4.6 PRACTICAL IMPROVEMENTS

For more precise predictions of shading contributions with arbitrary BRDFs, there is a body of recent research to draw on, such as accurate analytic estimation of the unshadowed shading via *linearly transformed cosines* (LTCs) for area lights [16] and linear lights [17]. Optimal light sampling strategies on the level of individual light sources are a topic of ongoing research (and with it, analytic integration to obtain corresponding PDFs), with notable recent advancements for area lights [23] and sphere lights [24].

47.5 REDUCING VARIANCE WITH PSEUDO-MARGINAL MIS

A typical issue of light sampling algorithms is that, with highly reflective materials, predicting the shading contribution of lights is difficult to impossible (depending on visibility): the shading contribution

$f(\mathbf{y}) = f_r\left(\mathbf{o}, \mathbf{x}, \mathbf{i}\right)\ \mathbf{n}^T\mathbf{i}\ V(\mathbf{x}, \mathbf{y})\ L_l(\mathbf{x}, \mathbf{y})$ (see Equation 47.1) for such BSDFs f_r may be very different even for individual points $\mathbf{y}$ on the same light $\mathcal{L}_l$, making its shading $\int_{\mathcal{L}_l} f(\mathbf{y})\,\mathrm{d}\mathbf{y}$ dependent on the precise shape of the light source and its visibility. In production rendering, such shading is often left to other sampling techniques than light sampling, such as finding lights by simple ray tracing instead: rays into relevant directions are sampled proportionally to the scattering profile of the material. In this section, we adapt this strategy into the context of real-time light sampling in Q2VKPT.

47.5.1 MULTIPLE IMPORTANCE SAMPLING

The decision where which sampling technique works best and should therefore be trusted most can be made by the heuristics of multiple importance sampling [31]. The key observation of MIS is that in MC estimation, for each sampled point $\mathbf{y}$ that contributes light $f(\mathbf{y})$, we can decide individually what fraction $w_1(\mathbf{y})$ of its contribution should be estimated by one technique such as light sampling and what fraction $w_2(\mathbf{y})$ by another technique like tracing with BSDF scattering rays. The two sampling techniques will generate the same points $\mathbf{y}$ with different probability distributions, e.g., $p_1(\mathbf{y})$ for light sampling and $p_2(\mathbf{y})$ for BSDF sampling, while the weighted sum of their samples will still converge to the shading we want to compute:

$$\mathbf{E}_{\substack{p_2(\mathbf{y}_2)\\ \cdot p_1(\mathbf{y}_1)}}\left[\sum_{i=1}^{2} w_i(\mathbf{y}_i)\frac{f(\mathbf{y}_i)}{p_i(\mathbf{y}_i)}\right] = \int_{\mathcal{L}}\int_{\mathcal{L}} \sum_{i=1}^{2} w_i(\mathbf{y}_i) f(\mathbf{y}_i) \prod_{j\neq i}^{2} \underbrace{p_j(\mathbf{y}_j)\,\mathrm{d}\mathbf{y}_j}_{\int_{\mathcal{L}} p_j(\mathbf{y}_j|\mathrm{U})\,\mathrm{d}\mathbf{y}_j=1} \mathrm{d}\mathbf{y}_i \tag{47.11}$$

$$= \sum_{i=1}^{2} \int_{\mathcal{L}} w_i(\mathbf{y}_i) f(\mathbf{y}_i)\,\mathrm{d}\mathbf{y}_i = \int_{\mathcal{L}} \underbrace{\left(\sum_{i=1}^{2} w_i(\mathbf{y})\right)}_{=1} f(\mathbf{y})\,\mathrm{d}\mathbf{y}. \tag{47.12}$$

A simple weighting heuristic that retains the good parts of light sampling with $p_1(\mathbf{y})$ and BSDF sampling with $p_2(\mathbf{y})$ is the balance heuristic [31] with weights $w_i(\mathbf{y}) = p_i(\mathbf{y}) / \sum_j p_j(\mathbf{y})$, conveniently resulting in the same MC estimates regardless of the sampling strategy used to generate $\mathbf{y}$:

$$w_i(\mathbf{y})\frac{f(\mathbf{y})}{p_i(\mathbf{y})} = \frac{f(\mathbf{y})}{\sum_j p_j(\mathbf{y})}. \tag{47.13}$$

Note that the resulting denominator in fact corresponds to the marginal probability distribution of using both sampling techniques in parallel.

47.5.2 STOCHASTIC MULTIPLE IMPORTANCE SAMPLING

In the following, we adapt MIS to add BSDF sampling along with the light sampling in Q2VKPT. This significantly reduces noise and denoiser artifacts for glossy materials in some hard cases (see Figure 47-5 in Section 47.7). However, an interesting challenge arises when we want to apply MIS to our stochastic subset sampling strategy: in order to compute the denominator $p_1(\mathbf{y}) + p_2(\mathbf{y})$ for our MIS-weighted shading estimates, we would need to know the marginal probability density of sampling from all subsets J_s (see Section 47.4) rather than from one subset only. For our previous strided subset selection scheme, if we found a light by ray tracing with PDF $p_2(\mathbf{y})$, for computing $p_1(\mathbf{y})$ we would at least have to look at all the lights in the subset of that light source, also.

To avoid this overhead, we construct a generalized stochastic variant of MIS that allows us to circumvent marginalizing with respect to all possible ways of sampling $\mathbf{y}$. Such generalizations were recently discussed as approximations of continuous MIS [34] and as generalized (discrete) MIS [12]. As a generalized form of generating stochastic subsets of all relevant lights J, let I_U denote subsets that are constructed using any number of uniform random variables $U = (u_1, u_2, ..., u_R) \in \mathcal{U}, \mathcal{U} = [0, 1]^R$. The balance heuristic can be applied to only the sampling technique resulting from the stochastic subset:

$$\mathbf{E}_{p_2(\mathbf{y}_2|U)\,p_1(\mathbf{y}_1|U)\,p(U)}\left[\sum_{i=1}^{2}\frac{f(\mathbf{y}_i)}{p_1(\mathbf{y}_i|U)+p_2(\mathbf{y}_i)}\right]$$

$$= \int_{\mathcal{U}}\int_{\mathcal{L}}\int_{\mathcal{L}}\sum_{i=1}^{2}\frac{f(\mathbf{y}_i)}{p_1(\mathbf{y}_i|U)+p_2(\mathbf{y}_i)}p_i(\mathbf{y}_i|U)\underbrace{p(U)}_{=1}\prod_{j\neq i}^{2}\underbrace{p_j(\mathbf{y}_j|U)\,\mathrm{d}\mathbf{y}_j}_{\int_{\mathcal{L}}p_j(\mathbf{y}_j|U)\,\mathrm{d}\mathbf{y}_j=1}\mathrm{d}\mathbf{y}_i\,\mathrm{d}U \tag{47.14}$$

$$= \int_{\mathcal{U}}\sum_{i=1}^{2}\int_{\mathcal{L}}\frac{f(\mathbf{y}_i)}{p_1(\mathbf{y}_i|U)+p_2(\mathbf{y}_i)}\underbrace{p_i(\mathbf{y}_i|U)}_{\substack{\text{note:}\\ p_2(\mathbf{y}|U)=p_2(\mathbf{y})}}\mathrm{d}\mathbf{y}_i\,\mathrm{d}U = \int_{\mathcal{L}}\left(\underbrace{\int_{\mathcal{U}}\mathrm{d}U}_{=1}\right)f(\mathbf{y})\,\mathrm{d}\mathbf{y}. \tag{47.15}$$

However, it is important to note that such naive randomization of MIS may render its weighting ineffective for variance reduction: whenever BSDF sampling hits a light source that is not contained by the current stochastic subset I_U, the PDF $p_1(\mathbf{y}|U)$ becomes zero and MIS is effectively inactive.

47.5.3 PSEUDO-MARGINAL MIS

For good variance reduction performance, we need to ensure that any stochastic subset we select also helps with variance reduction for any lights

hit by BSDF samples. Our key insight is that we do not have to choose between full marginalization or full randomization of MIS weights. In particular, it is also possible to marginalize with respect to one random variable only. For example, if in the subset I_U each selected light source was chosen with one random variable u_j, then we can marginalize with respect to u_1, while reusing all the information of light sources chosen with the other random variables $u_{j\neq 1}$:

$$\mathbf{E}_{p_2(\mathbf{y}_2|\mathsf{U})\,p_1(\mathbf{y}_1|\mathsf{U})\,p(\mathsf{U})}\left[\sum_{i=1}^{2}\frac{f(\mathbf{y}_i)}{\int_{[0,1]}p_1(\mathbf{y}_i|\mathsf{U})\,\mathrm{d}u_1+p_2(\mathbf{y}_i)}\right]$$

$$=\int_{\mathcal{U}}\sum_{i=1}^{2}\int_{\mathcal{L}}\underbrace{\frac{f(\mathbf{y}_i)}{\int_{[0,1]}p_1(\mathbf{y}_i|\mathsf{U})\,\mathrm{d}u_1+p_2(\mathbf{y}_i)}}_{\text{note: does not depend on } u_1}p_i(\mathbf{y}_i|\mathsf{U})\,\mathrm{d}\mathbf{y}_i\,\mathrm{d}\mathsf{U} \tag{47.16}$$

(see Equations 47.14 and 47.15)

$$=\int_{[0,1]^{R-1}}\sum_{i=1}^{2}\int_{\mathcal{L}}\frac{f(\mathbf{y}_i)}{\int_{[0,1]}p_1(\mathbf{y}_i|\mathsf{U})\,\mathrm{d}u_1+p_2(\mathbf{y}_i)}\underbrace{\left(\int_{[0,1]}p_i(\mathbf{y}_i|\mathsf{U})\,\mathrm{d}u_1\right)}_{\substack{\text{note:}\\ \int_{[0,1]}p_2(\mathbf{y}|\mathsf{U})\,\mathrm{d}u_1=p_2(\mathbf{y})}}\mathrm{d}\mathbf{y}_i\,\mathrm{d}(u_2,\ldots) \tag{47.17}$$

$$=\int_{\mathcal{L}}\underbrace{\left(\int_{[0,1]^{R-1}}\mathrm{d}(u_2,\ldots)\right)}_{=1}f(\mathbf{y})\,\mathrm{d}\mathbf{y}. \tag{47.18}$$

The marginalization of $p_1(\mathbf{y}|\mathsf{U})$ with respect to u_1 is simple, as the resulting PDF only depends on whether or not u_1 adds l to I_U for the respective $\mathcal{L}_l \ni \mathbf{y}$. The probability of the respective events is multiplied by the respective values of $p_1(\mathbf{y}|\mathsf{U})$. With this, we can ensure that the marginal density $\int_{[0,1]}p_1(\mathbf{y}|\mathsf{U})\,\mathrm{d}u_1$ is nonzero for any $\mathbf{y}\in\mathcal{L}_l$ found by BSDF sampling, as long as we ensure that any l can be selected with u_1.

47.5.4 STRATIFIED PSEUDO-MARGINAL MIS

Sampling light sources in the stochastic subsets I_U completely independently has disadvantages, because the same light source may unnecessarily be chosen twice. We can simplify computing the marginalization of $p(\mathbf{y}|\mathsf{U})$ by stratification, and even increase the resulting probability density reliably: We define a stratified stochastic subset J_U, based on a list of random variables $\mathsf{U}=(u_1,\ldots,u_R)$, each random variable u_i independent and uniform, as

$$\mathsf{J}_\mathsf{U} := \mathsf{J}\left[iS+\lfloor u_i S\rfloor \,|\, i\in\mathbb{N}_0\right], \tag{47.19}$$

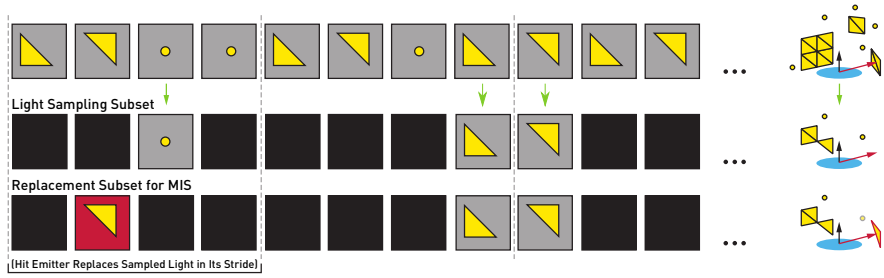


Figure 47-3. *To enable MIS, stochastic light subsets are randomized independently within each stride. This allows us to compute additional sampling weights for emitters hit by BSDF rays: We can replace any light selected in the corresponding subset stride by the hit emitter, and thus obtain another representative stochastic subset for that emitter with minimal additional information.*

that is, each random variable u_i selects an index from the list of all relevant lights J within a stride of S light indices. This is illustrated in Figure 47-3. For a given $\mathbf{y}$, we marginalize with respect to the random variable u_j that is responsible for selecting $\mathcal{L}_l$, where $y \in \mathcal{L}_k$. The only case in which $p_1(\mathbf{y}|\mathrm{U})$ is nonzero is when $\mathrm{J_U}$ contains within the respective stride $[jS, (j+1)S)$ in the list J:

$$\int_{[0,1]} p_1(\mathbf{y}_i|\mathrm{U})\,\mathrm{d}u_j = \frac{1}{S}\,p_1(\mathbf{y}_i|\mathrm{U}\text{ setting }u_j\text{ such that }l \in \mathrm{J_U}). \tag{47.20}$$

Note that with this we effectively perform different marginalizations for different $\mathbf{y} \in \mathcal{L}$, but the derivation in Equation 47.18 still works after the integral over $\mathcal{L}$ is split into respective regions, each cancelling out a different marginalized PDF.

47.6 STOCHASTIC LIGHT SUBSET MIS

To implement the idea of pseudo-marginal MIS, we need to change our way of stochastically sampling subsets, such that for each subset selected in light sampling, it becomes easy to construct a similar, equally probable subset that is guaranteed to contain a specific emitter hit by ray tracing. This can be achieved by independently randomizing the subset selection process in each stride of the light list (see Equation 47.19), such that for every subset, there exists another equally probable subset that replaces one light by the hit emitter, within the respective light list stride (see Figure 47-3). In Listing 47-3, we identify this stride during the light sampling process and additionally keep track of the total weight of all lights in the subset except for the one in the stride of the hit emitter.

Listing 47-3. *Light sampling with information for MIS weighting of BSDF-ray light hits, using one random variable ξ_1.*

```
S = ⌈(light_list_end-light_list_begin) / MAX_SUBSET_LEN⌉
subset_stride = S

selected = (light_idx: -1, mass: 0)
total_weights = 0

// Additional information and randomization for MIS
hit_caught = not hit_emitter
other_weights = 0 // Weights excluding hit_emitter stride
pending_weight = 0
FastRng small_rnd(seed: ξ1, mod: S)

light_offset = light_list_begin
for (i = 0; i < MAX_SUBSET_LEN; ++i) {
    light_idx = light_offset + small_rnd()
    if (light_idx >= light_list_end) {
        // Detect if hit_emitter is in current stride.
        hit_caught ||=   light_offset < light_list_end
                      && light_pointers[light_offset] <= hit_emitter
        break
    }
    w = light_contrib(v, p, n, light_pointers[light_idx])
    // Accumulate all weights outside hit_emitter's stride.
    wo = w
    if (not hit_caught && hit_emitter <= light_pointers[light_idx]) {
        // Is the emitter in this or the last stride?
        if (light_pointers[light_offset] <= hit_emitter)
            wo = 0 // This stride
        else
            pending_weight = 0 // Last stride
        hit_caught = true // Found hit_emitter
    }
    other_weights += pending_weight
    pending_weight = wo

    if (w > 0) {
        τ = total_weights / (total_weights+w); total_weights += w
        if (ξ1 < τ) { ξ1 /= τ }
        else { selected = (light_pointers[light_idx], w); ξ1 = (ξ1-τ) / (1-τ) }
        ξ1 = clamp(ξ1, 0, MAX_BELOW_ONE)
    }
    light_offset += subset_stride
}
// Compute pseudo-marginal probability of sampling hit_emitter.
if (hit_caught)
    other_weights += pending_weight
hit_w = light_contrib(v, p, n, hit_emitter)
hit_probability = hit_w / ((other_weights + hit_w) * S)

probability = selected.mass / (total_weights * S)
return (selected.light, probability, hit_probability)
```

47.6.1 INDEPENDENTLY SELECTING LIGHTS PER STRIDE

In order to independently sample one light per light list stride, we use a fast linear congruential generator (`FastRng` in Listing 47-3) to generate integers in $[0, S - 1]$ in each iteration of the loop (line 15). Note that simply using one floating-point random number, as in the incremental sampling scheme of the previous section, is not generally feasible because the number of random bits may be exceeded even for lower stride widths and counts.

47.6.2 IDENTIFYING THE STRIDE OF HIT EMITTERS

Once a ray hits an emitter by chance, e.g., by BSDF sampling, we need be able to identify its subset stride in the light sampling process and replace the corresponding importance sampling weight therein by that of the hit emitter. It is nontrivial to keep track of the offsets of the emitters in all light lists of the scene because, as in any real-world application, there are likely many light lists adapted to capture the relevant illumination for different shading points.

We tackle this problem by enforcing an order for emitters in light lists, i.e., we introduce an (arbitrary) order defined by the memory location of each emitter. Once all light lists are sorted in ascending order, we can infer the strides that potentially contain a given hit emitter `hit_emitter` during light sampling by simple pointer comparisons with other lights in the subset (Listing 47-3, lines 18 and 23). This allows us to obtain the weight sum `other_weights` that only contains the weights of lights in the current subset that do not overlap with the stride of the hit emitter.

Finally, we are able to compute the pseudo-marginal probability required for MIS at the hit emitter (line 48), knowing the total weight `hit_w` + `other_weights` of the corresponding alternative subset (compare to Figure 47-3).

47.7 RESULTS AND DISCUSSION

All variants of our method run in real time on an NVIDIA GeForce RTX 2070, at a resolution of 1920×1080 at 11–17 ms of frame time (including denoising). We implemented our method in the open source Q2VKPT project [25], which is based on Vulkan using the hardware-accelerated ray tracing extension. We show unfiltered one-sample-per-pixel path tracer outputs with one indirect light bounce and outputs that were filtered using a variant of spatiotemporal filtering [27].

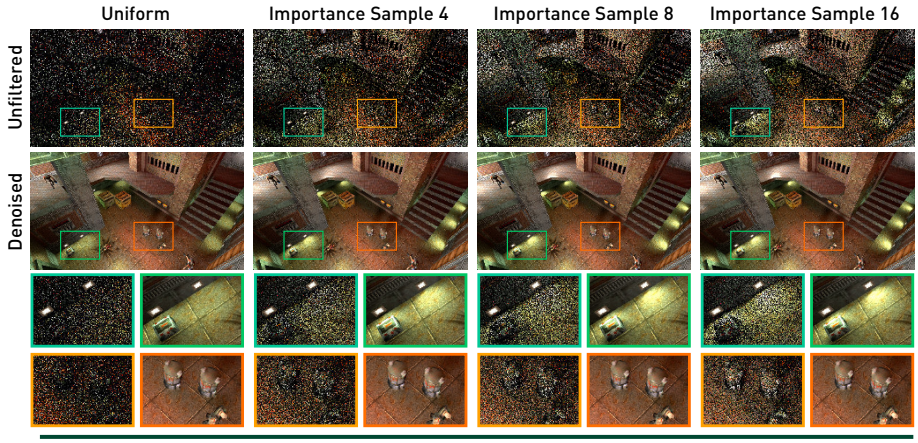


Figure 47-4. *Light importance sampling with various stochastic subset sizes, at one sample per pixel, unfiltered (top) and denoised with ASVGF [27] (bottom). The quality of light sampling increases with the size of the stochastic subsets, balancing more samples with respect to other lights. With uniform sampling, even clamped path tracing produces high variance, as lights are randomly far away or nearby, in or out of the focus of specular highlights. This leads to unstable results, even after temporal filtering, causing flickering in motion. Importance sampling in stochastic subsets is effective in reducing variance, bringing results closer to a converged result even at one sample per pixel.*

47.7.1 RUNTIMES

The path tracer runs at 7–9 ms per frame with one indirect bounce and a stochastic light subset size of 8. Randomizing the subset by choosing individual offsets per stride as in Section 47.6 adds about 0.1 ms compared to the randomization with regular intervals as in Section 47.4. Enabling MIS adds another 1–1.5 ms, as it requires computing and tracking a few additional quantities and additional emitter texture accesses for the hit points of BSDF-sampled rays.

47.7.2 SUBSET SIZES

As shown in Figure 47-4, subset sizes affect the quality of results, demonstrating how good importance sampling improves the performance of a Monte Carlo path tracer. Especially for smaller subset sizes, the resulting noise is reduced significantly with every additional light in the subset. For Q2VKPT, we chose a subset size of 8 as a trade-off between sampling quality and performance because it gave decent results that looked stable after denoising. In our (unprofiled, not thoroughly optimized) implementation, going from a subset size of 8 to 16 adds 1 ms, going to 32 another 1.5 ms, and up to 64 another 3 ms.

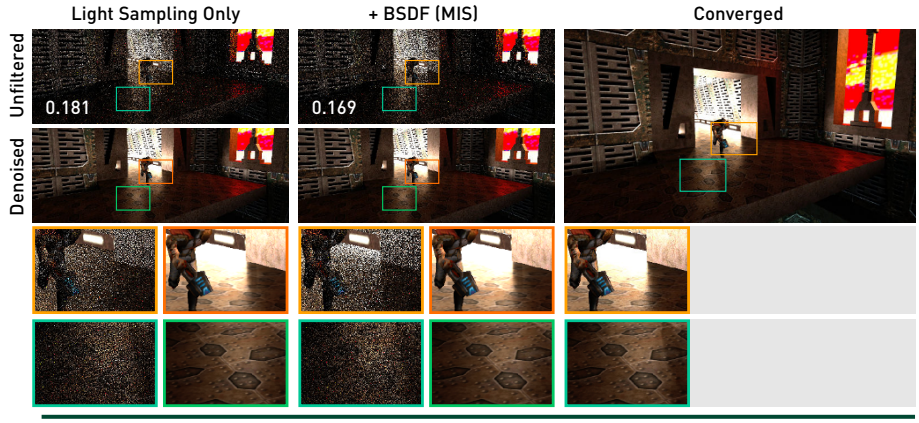


Figure 47-5. *Our light sampling with and without MIS-weighted BSDF samples, at one sample per pixel, unfiltered (top) and denoised (bottom). As expected, the RMSE numbers (top row) decrease with our pseudo-marginal MIS. Top inset row: MIS makes up for cases where our light sampling alone underestimates contributions and samples the floor lights too rarely. Without MIS, we would need to clamp these samples to obtain robust denoising results under motion. Bottom inset row: unfortunately, the additional randomization, required for our pseudo-marginal MIS, destroys the stratification of pixel error in screen space. Under motion, this results in slightly less stable denoising results.*

47.7.3 MULTIPLE IMPORTANCE SAMPLING

The impact of MIS (with emitters selected by light sampling and emitters hit by BSDF sampling) on our results is shown in Figures 47-5 and 47-6. There are two noticeable cases where MIS with BSDF sampling makes up for typical shortcomings of light sampling approaches. In the first case, around the center of a specular highlight, when the corresponding emitter is nearby, different points on the emitter can have very different contributions to the shaded point. Here, our approximate product importance sampling fails to predict the exact contribution of the emitter as a whole, and thus assigns misleading importance weights to it. The other typical failure case affects points in direct proximity of emitters, where the inverse squared distance law of light falloff results in individual emitter points contributing unbounded sample values for light sampling. In both cases, MIS identifies BSDF sampling as the better sampling technique, because its probability density covers both specular peaks and nearby points well. As a result, light samples are downweighted and BSDF samples upweighted in these locations, making up for the shortcomings of light sampling and ensuring stable shading estimates.

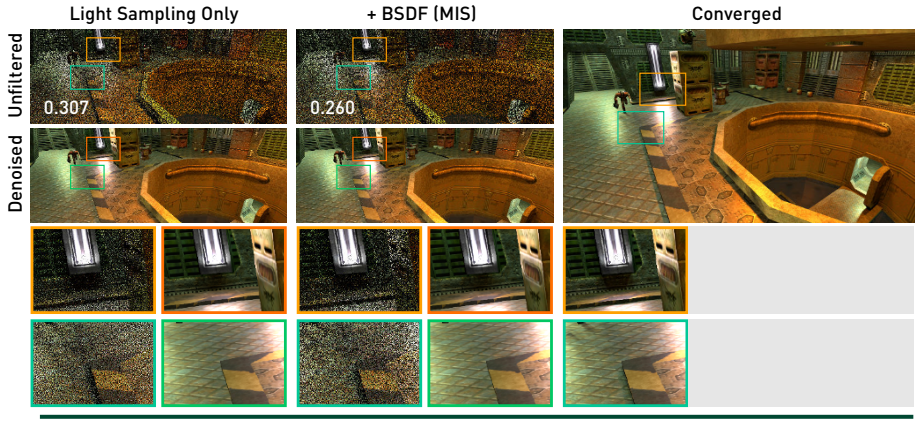


Figure 47-6. *Our light sampling with and without MIS-weighted BSDF samples, at one sample per pixel, unfiltered (top) and denoised (bottom). Root mean square error numbers (RMSE) are inset. See Figure 47-5 for discussion.*

47.7.4 STRATIFICATION

Our stratified subset sampling strategy without MIS (Section 47.4) works well with stratified random variables (in our case blue noise points), as is visible in Figure 47-7 (left). Unfortunately, this is no longer the case when we add pseudo-marginal MIS (Section 47.6). The required additional randomization destroys the correspondence between convex random variable intervals and light sources, destroying the positive effect of stratification. In Figure 47-7 (middle) we can see the noticeable clumping of white noise that we would expect from independent random variables. Regrettably, the resulting higher variance in the denoised output in most cases cancels out any benefits we received from implementing our MIS. Therefore, it is likely a better idea to use the variant without MIS and instead improve on the product importance estimation in the future. To verify that the additional randomization is the problem, Figure 47-7 (right) shows a biased variant of MIS that fixes the stratification problems, at the cost of an unpredictable systematic error in the rendered output.

47.8 CONCLUSIONS

Variance reduction techniques like importance sampling and MIS are crucial tools not only in offline rendering, but even more so in upcoming real-time path tracing applications with low sample counts. For Q2VKPT, we found that

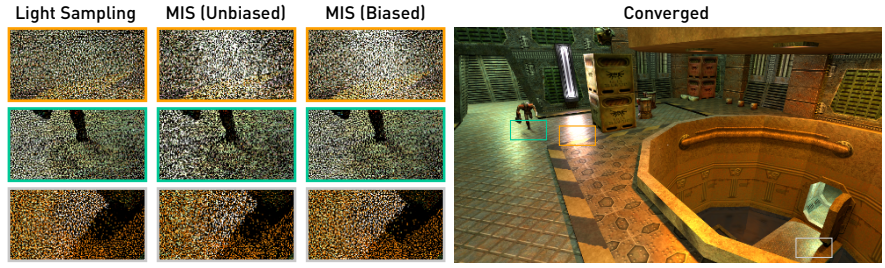


Figure 47-7. *Left: our stratified subset sampling works well with blue noise points, and the error is nicely diffused in screen space. Middle: unfortunately, our unbiased pseudo-marginal MIS cannot profit from stratified random variables. In the noise, we can see the typical clumping of white noise, which we would expect from independent random variables. Using pure light sampling and developing more accurate product importance sampling strategies is therefore likely a more promising avenue for future work. Right column: we can verify that the additional randomization required by pseudo-marginal MIS is the culprit. Disabling it leads to a biased result, but recovers the stratification effects in screen space. Note that this will only look similar as long as the lights contained in each stratified subset are sufficiently representative of the full illumination. In practice, the unpredictable bias is likely undesirable.*

even a simplistic resampling method, working with small stochastic subsets of the full scene illumination, can bring tremendous quality improvements. The feasibility of real-time Monte Carlo rendering with denoising depends on such strategies of variance control, as they directly affect the amount of reliable information in the framebuffer that can be used to reconstruct correct and stable results. More recent developments than our approaches like ReSTIR [4] and real-time path guiding [11] make good steps toward even more robust techniques, which in some cases barely require denoising at all. We are hopeful that these techniques can be combined in their robustness and generality, leading to even more reliable and versatile rendering algorithms in the future.

Finally, in this chapter we explored the benefits and challenges of adding MIS to the light sampling technique that was released with Q2VKPT. We proposed pseudo-marginal stochastic MIS to improve some difficult corner cases where our approximate product importance sampling fails. In our use case, we found its usefulness to be limited due to the resulting loss of stratification benefits. It will likely be made obsolete by more sophisticated real-time light sampling strategies and future product importance sampling techniques. Nevertheless, we hope that our look into pseudo-marginal stochastic MIS may serve as an inspiration for the design of variance reduction techniques in other use cases where time for the exhaustive evaluation of alternative PDFs is limited.

ACKNOWLEDGMENTS

We thank Christoph Schied, Addis Dittebrandt, and Christoph Peters for many insightful discussions around light sampling and the Q2VKPT project. Q2VKPT was built by Christoph Schied [25], and we thank him for creating the opportunity of contributing to the project, with additional contributions coming from Johannes Hanika, Addis Dittebrandt, Florian Reibold, Stephan Bergmann, Emanuel Schrade, Alisa Jung, and Christoph Peters.

REFERENCES

[1] Arvo, J. Stratified sampling of spherical triangles. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, pages 437–438, 1995. DOI: 10.1145/218380.218500.

[2] Bikker, J. *Ray Tracing for Real-Time Games*. PhD thesis, Technische Universiteit Delft, 2012.

[3] Bikker, J. Real-time ray tracing through the eyes of a game developer. In *IEEE Symposium on Interactive Ray Tracing*, page 1, 2007. DOI: 10.1109/RT.2007.4342583.

[4] Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A., and Jarosz, W. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 39(4):148:1–148:17, 2020. DOI: 10.1145/3386569.3392481.

[5] Burke, D., Ghosh, A., and Heidrich, W. Bidirectional importance sampling for direct illumination. In *Proceedings of the Sixteenth Eurographics Conference on Rendering Techniques*, pages 147–156, 2005.

[6] Burley, B., Adler, D., Chiang, M. J.-Y., Driskill, H., Habel, R., Kelly, P., Kutz, P., Li, Y. K., and Teece, D. The design and evolution of Disney's Hyperion renderer. *ACM Transactions on Graphics*, 37(3):33:1–33:22, 2018. DOI: 10.1145/3182159.

[7] Chaitanya, C. R. A., Kaplanyan, A. S., Schied, C., Salvi, M., Lefohn, A., Nowrouzezahrai, D., and Aila, T. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics*, 36(4):98:1–98:12, 2017. DOI: 10.1145/3072959.3073601.

[8] Chao, M.-T. A general purpose unequal probability sampling plan. *Biometrika*, 69(3):653–656, 1982.

[9] Christensen, P., Fong, J., Shade, J., Wooten, W., Schubert, B., Kensler, A., Friedman, S., Kilpatrick, C., Ramshaw, C., Bannister, M., et al. RenderMan: An advanced path-tracing architecture for movie rendering. *ACM Transactions on Graphics*, 37(3):30:1–30:21, 2018. DOI: 10.1145/3182162.

[10] Conty Estevez, A. and Kulla, C. Importance sampling of many lights with adaptive tree splitting. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(2):25:1–25:17, 2018. DOI: 10.1145/3233305.

[11] Dittebrandt, A., Hanika, J., and Dachsbacher, C. Temporal sample reuse for next event estimation and path guiding for real-time path tracing. In *Eurographics Symposium on Rendering—DL-Only Track*, pages 39–51, 2020. DOI: 10.2312/sr.20201135.

[12] Elvira, V., Martino, L., Luengo, D., and Bugallo, M. F. Generalized multiple importance sampling. *Statistical Science*, 34(1):129–155, 2019. DOI: 10.1214/18-STS668.

[13] Fascione, L., Hanika, J., Leone, M., Droske, M., Schwarzhaupt, J., Davidovič, T., Weidlich, A., and Meng, J. Manuka: A batch-shading architecture for spectral path tracing in movie production. *ACM Transactions on Graphics*, 37(3):31:1–31:18, 2018. DOI: 10.1145/3182161.

[14] Georgiev, I., Ize, T., Farnsworth, M., Montoya-Vozmediano, R., King, A., Lommel, B. V., Jimenez, A., Anson, O., Ogaki, S., Johnston, E., Herubel, A., Russell, D., Servant, F., and Fajardo, M. Arnold: A brute-force production path tracer. *ACM Transactions on Graphics*, 37(3):32:1–32:12, 2018. DOI: 10.1145/3182160.

[15] Georgiev, I., Krivanek, J., Popov, S., and Slusallek, P. Importance caching for complex illumination. *Computer Graphics Forum*, 31(2pt3):701–710, 2012. DOI: 10.1111/j.1467-8659.2012.03049.x.

[16] Heitz, E., Dupuy, J., Hill, S., and Neubelt, D. Real-time polygonal-light shading with linearly transformed cosines. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 35(4):41:1–41:8, July 2016. DOI: 10.1145/2897824.2925895.

[17] Heitz, E. and Hill, S. Linear-light shading with linearly transformed cosines. In W. Engel, editor, *GPU Zen Advanced Rendering Techniques*, pages 137–162. Black Cat Publishing, 2017.

[18] Keller, A., Wächter, C., Raab, M., Seibert, D., van Antwerpen, D., Korndörfer, J., and Kettner, L. The iray light transport simulation and rendering system. In *ACM SIGGRAPH 2017 Talks*, 34:1–34:2, 2017. DOI: 10.1145/3084363.3085050.

[19] Lin, D. and Yuksel, C. Real-time stochastic lightcuts. *Proceedings of the ACM on Computer Graphics and Interactive Techniques (Proceedings of I3D 2020)*, 3(1):5:1–5:18, 2020. DOI: 10.1145/3384543.

[20] Moreau, P. and Clarberg, P. Importance sampling of many lights on the GPU. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 255–283. Apress, 2019.

[21] Moreau, P., Pharr, M., and Clarberg, P. Dynamic many-light sampling for real-time ray tracing. In *High-Performance Graphics 2019—Short Papers*, 2019. DOI: 10.2312/hpg.20191191.

[22] Müller, T., Gross, M., and Novák, J. Practical path guiding for efficient light-transport simulation. *Computer Graphics Forum*, 36(4):91–100, 2017. DOI: 10.1111/cgf.13227.

[23] Peters, C. BRDF importance sampling for polygonal lights. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 38(4), July 2021. To appear.

[24] Peters, C. and Dachsbacher, C. Sampling projected spherical caps in real time. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 2(1):1:1–1:16, June 2019. DOI: 10.1145/3320282.

[25] Schied, C. Q2VKPT. http://brechpunkt.de/q2vkpt/, with source code at https://github.com/cschied/q2vkpt, 2019.

[26] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*. ACM, 2017.

[27] Schied, C., Peters, C., and Dachsbacher, C. Gradient estimation for real-time adaptive temporal filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(2):24:1–24:16, 2018. DOI: 10.1145/3233301.

[28] Schmittler, J., Pohl, D., Dahmen, T., Vogelgesang, C., and Slusallek, P. Realtime ray tracing for current and future games. In *ACM SIGGRAPH 2005 Courses*, 23–es, 2005. DOI: 10.1145/1198555.1198762.

[29] Shirley, P., Wang, C., and Zimmerman, K. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics*, 15(1):1–36, 1996. DOI: 10.1145/226150.226151.

[30] Talbot, J., Cline, D., and Egbert, P. Importance resampling for global illumination. In *Proceedings of the Eurographics Workshop on Rendering*, pages 139–146, 2005. DOI: 10.2312/EGWR/EGSR05/139-146.

[31] Veach, E. and Guibas, L. J. Optimally combining sampling techniques for Monte Carlo rendering. In pages 419–428, 1995.

[32] Vorba, J., Karlík, O., Šik, M., Ritschel, T., and Křivánek, J. On-line learning of parametric mixture models for light transport simulation. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 33(4):101:1–101:11, 2014. DOI: 10.1145/2601097.2601203.

[33] Walter, B., Fernandez, S., Arbree, A., Bala, K., Donikian, M., and Greenberg, D. P. Lightcuts: A scalable approach to illumination. *ACM Transactions on Graphics*, 24(3):1098––1107, 2005. DOI: 10.1145/1073204.1073318.

[34] West, R., Georgiev, I., Gruson, A., and Hachisuka, T. Continuous multiple importance sampling. *ACM Transactions on Graphics*, 39(4):136:1–136:12, 2020. DOI: 10.1145/3386569.3392436.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 48

RAY TRACING IN FORTNITE

Patrick Kelly,[1] Yuriy O'Donnell,[1] Kenzo ter Elst,[1] Juan Cañada,[1] and Evan Hart[2]
[1]Epic Games
[2]NVIDIA

ABSTRACT

In this chapter we describe implementation details of some of the Unreal Engine 4 ray tracing effects that shipped in *Fortnite* Season 15. In particular, we dive deeply into ray traced reflections and global illumination. This includes goals, practical considerations, challenges, and optimization techniques.

48.1 INTRODUCTION

DirectX Raytracing (DXR) in Unreal Engine 4 (UE4) was showcased for the first time at Game Developers Conference 2018 in *Reflections*, a Lucasfilm *Star Wars* short movie made in collaboration with ILMxLAB and NVIDIA. A year after that, the first implementation of ray tracing in UE4 was released in



Figure 48-1. *Fortnite rendered with ray tracing effects.*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_48

UE 4.22. Initially, the focus of the engineering team was set on feature completion and stability rather than on performance for real-time graphics applications. For this reason, during the first year and a half, ray tracing adoption was significantly higher in enterprise applications for architecture, automotive, or film rather than in games. A detailed description of how ray tracing was implemented in UE4 can be found in the chapter "Cinematic Rendering in UE4 with Real-Time Ray Tracing and Denoising" in the first volume of *Ray Tracing Gems* [10].

The feasibility of ray tracing for games increased quickly due to the vast improvements happening both in hardware and software since 2018. UE4 licensees started to include some ray tracing effects in games and other applications that demand high frame rates. Early in 2020, Unreal Engine ray tracing was transitioning from beta stage to production, and the engineering team decided it was the right time to battle-test it under challenging conditions. *Fortnite* was perfect for this task, not only due to its massive scale but also because it possessed many other characteristics that make it difficult to implement in-game ray tracing. Namely, the content creation pipeline is very well defined and changing it to embrace ray tracing was not an option. Moreover, content updates happen very often so it is not possible to adjust parameters to make specific versions look good, but any modification should be relatively permanent and behave correctly in future updates.

This chapter explains how the ray tracing team at Epic Games implemented ray tracing techniques in *Fortnite* Season 15. We describe the challenges, the trade-offs, and the solutions found.

48.2 GOALS

The main goal of the project was to ship ray tracing in *Fortnite*, both to improve the game's visuals and to battle-proof the UE4 ray tracing technology. From the technical perspective, the initial objective was to run the game on a system with an 8-core CPU (i7-7000 series or equivalent) and an NVIDIA 2080 Ti graphics card with the following requirements:

- Frame rate: 60 FPS.
- Resolution: 1080p.
- Ray tracing effects: shadows, ambient occlusion, and reflections.

As described in the following sections, improvements that happened during the project helped to achieve more ambitious goals. Novel developments in

ray traced reflections, global illumination, and denoising, plus the integration of NVIDIA's Deep Learning Super Sampling (DLSS), made it possible to target higher resolutions and more sophisticated lighting effects such as ray traced global illumination (RTGI).

From the art and content creation point of view, the team was not aiming for a dramatic change in the look, but the goal was to achieve specific improvements on key lighting effects that could make the visuals more pleasant by removing some artifacts introduced by screen-space effects. *Fortnite* is a non-photorealistic game, and the intention was to avoid an experience where ray tracing and rasterization looked too different.

48.3 CHALLENGES

From the performance side, initial tests showed both the CPU and GPU were far away from the initial performance goals when ray tracing was enabled. When running on the target hardware, CPU time was around 24 ms per frame on average, with some peaks of more than 30 ms. GPU performance was also far from the goal. While some scenarios were fast enough, others with more complex lighting were in the 30–40 ms per frame range. Some pathological cases with many dynamic geometries (such as trees) were extremely slow—on the order of 100 ms per frame.

Besides performance, there were other areas that presented interesting challenges. A remarkable one was the content creation pipeline. *Fortnite* manages an extraordinarily large amount of assets that are updated at very high frequency. Changing assets to look better in ray tracing was not possible because the overload on the content team would be unacceptable. For example, to improve the performance of ray traced reflections, the team considered adding a flag to set if an object was casting reflection rays or not. However, after further evaluation it was clear that such a solution would not scale. Any improvement had to work reasonably well *automatically* for all the existing and future content.

The look and feel of the game also presented a challenge. *Fortnite* does not have a photorealistic visual style. Most of the surfaces are highly diffuse. Reflections do not play an important role, except when water is present. Many talented artists and engineers have worked hard for years adjusting content and creating technology to make the game look good and to avoid artifacts that screen-space rasterized techniques produce. Making ray tracing shine

under these circumstances—without being able to change content—was a difficult task.

Another challenge worth mentioning was that the rendering API (known in Unreal Engine as the *Render Hardware Interface* (RHI)) used with ray tracing was not the default one. The default RHI in *Fortnite* is DirectX 11, but DXR runs on DirectX 12. This means that when enabling ray tracing, the game was exercising code paths that have been tested significantly less than the default ones. As expected, this revealed stability and performance issues that were not ray tracing specific but were critical to fix to make the game shippable. Improving the DirectX 12 RHI has been one of the most positive side effects of this initiative.

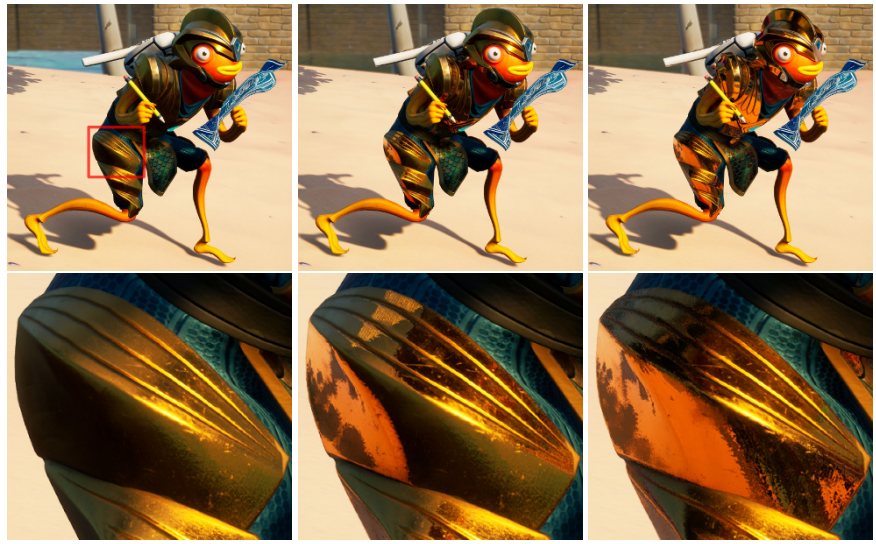
Lastly, another challenge came from a self-imposed limitation. All the technology developed for this project had to be included in the UE4 code base without any modification. Neither *Fortnite*, nor any other project developed at Epic Games, is allowed to customize the engine code. Though this can represent a problem in some cases, the long-term benefit of maintaining only one game engine overcomes any difficulties.

48.4 TECHNOLOGIES

This section describes the most important technologies that were improved or entirely developed during the project.

48.4.1 REFLECTIONS

Ray traced reflections have been a part of Unreal Engine for some time and were primarily used for cinematic rendering [10]; however, the *Fortnite* Season 15 release was the first time this effect was used in a game at Epic. Though our previous use cases required real-time performance, the goals were quite different from a game. A typical cinematic demo ran at 24 hertz at 1080p resolution and could require a high-end GPU. The *Fortnite* ray tracing target was *at least* 60 hertz at 1080p (4K with DLSS) on an NVIDIA 2080 Ti. Some optimizations and sacrifices had to be made to reach this goal. We made an experimental ray traced reflection implementation targeted specifically at games. It shared the main ideas from our original reflection shader, but shed most of the high-end rendering features such as multi-bounce reflections, translucent materials in reflections, physically based clear coat, and more. Figure 48-2 shows a comparison of the new ray traced reflections mode with screen-space reflections and with no reflections.



(a) *Reflections off* **(b)** *Screen-space reflections* **(c)** *Ray traced reflections*

Figure 48-2. *A comparison of reflection rendering modes.*

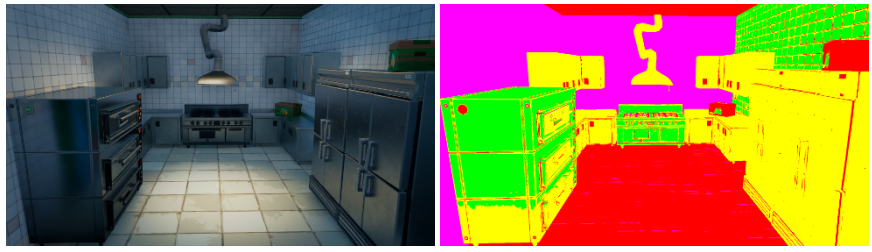
ALGORITHM OVERVIEW

The Unreal Engine reflection pipeline uses a sorted-deferred material evaluation scheme (Figure 48-3). First, reflection rays are generated based on G-buffer data and then traced to find the closest surfaces and their associated material IDs. The hit points are then sorted by material ID / shader. Finally, sorted hit points are used to dispatch another ray tracing pass that performs material evaluation and lighting using the full ray tracing pipeline state object (RTPSO).

The goal of this sorted pipeline is to improve material shader execution coherence (SIMD efficiency). Because reflection rays are randomized, pixels that are close in screen space will often generate rays that hit surfaces that are far apart—increasing the probability that they use different materials. If

Trace Rays → **Sort Hits by Material** → **Evaluate Materials** → **Lighting**

Figure 48-3. *The reflection pipeline using sorted-deferred material evaluation.*



(a) *Final reflections at the epic preset* **(b)** *Roughness threshold visualization*

Figure 48-4. *A visualization of different roughness threshold levels and corresponding GPU performance. Green: medium reflection quality preset, roughness ≤ 0.35, 0.7 ms. Yellow: high preset, roughness ≤ 0.55, 1.28 ms. Red: epic preset, roughness ≤ 0.75, 1.72 ms. Magenta: culled surfaces with roughness > 0.75, 2.09 ms. Timings given for NVIDIA RTX 3090 at* 1920 × 1080 *resolution.*

hit points with different materials end up in the same GPU wave,[1] the performance will drop roughly proportionally to the number of unique materials. Though theoretically a high-level ray tracing API such as DirectX Raytracing allows for automatic sorting to avoid this performance issue, in practice none of the drivers or hardware available at the time implemented this optimization. Implementing explicit sorting at the application level significantly improved performance [1], as described later.

RAY GENERATION

Reflection rays are generated using GGX distribution sampling based on G-buffer data [7]. To save GPU time, a roughness threshold is used to decide if a simple reflection environment map lookup can be used instead of tracing rays. The threshold value is mapped to the reflection quality parameter in *Fortnite* graphics options. We chose values 0.35, 0.55, and 0.75 for *medium*, *high*, and *epic* quality presets, respectively. All surfaces with roughness over 0.75 are culled, as performance cost is too high compared to the visual improvement. A special *low* preset also exists, disabling ray traced reflections on everything except water. Figure 48-4 shows a visualization of these quality presets and their GPU performance.

Because *Fortnite* content was not designed with ray traced reflection technology in mind, most of the assets were mastered for screen-space

[1] HLSL terminology is used in this chapter. *Wave* refers to a set of GPU threads executed simultaneously in a SIMD fashion. Similar concepts include Subgroup (Vulkan), Warp (NVIDIA), Wavefront (AMD).

Listing 48-1. *HLSL source code of the reflection roughness remapping function.*

```
float ApplySmoothBias(float Roughness, float SmoothBias)
{
    // SmoothStep-like function for Roughness values
    // lower than SmoothBias, original Roughness otherwise.
    float X = saturate(Roughness / SmoothBias);
    return Roughness * X * X * (3.0 - 2.0 * X);
}
```

reflections, which use pure mirror-like rays and therefore look quite sharp. A simple roughness threshold is used to cull screen-space reflections from most surfaces that are meant to appear rough/diffuse. Unfortunately, this means that physically based ray traced reflections appeared quite dull most of the time. Tweaking all materials manually was not an option due to the sheer amount of content in the game. Shown in Listing 48-1, an automatic solution was implemented that biases surface roughness during GGX sampling, making surfaces slightly shinier.

Shown in Figure 48-5, this remapping function pushes roughness values below some threshold closer to zero, but leaves higher values intact. This particular function was designed to preserve material roughness map

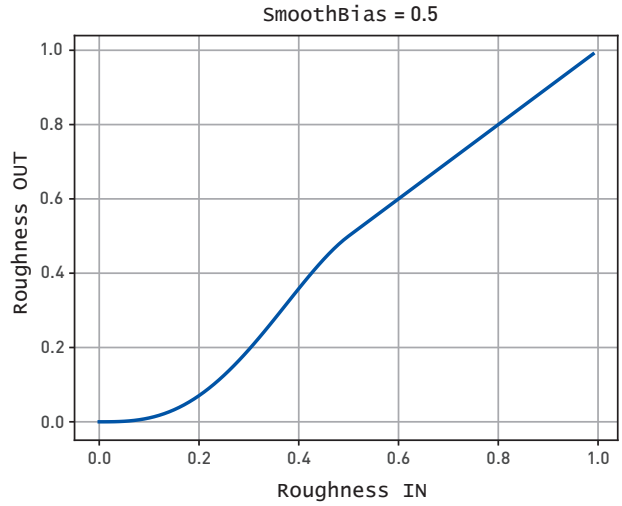


Figure 48-5. *A graph of the roughness remapping function for* `ApplySmoothBias(Roughness, 0.5)`*. This makes surfaces that are already smooth even smoother, while leaving rougher ones unchanged.*



(a) *Screen space* **(b)** *SmoothBias = 0.0* **(c)** *SmoothBias = 0.5* **(d)** *SmoothBias = 1.0*

Figure 48-6. *The visual effect of varying the smoothness bias.*

contributions without clipping, while remaining smooth over the full roughness range. A bias value of 0.5 was used in *Fortnite*, as it is a good compromise between the desired look and physical accuracy. As a small bonus, GPU performance was slightly improved in some scenes because mirror-like reflection rays are naturally more coherent (making them faster to trace). Figure 48-6 compares the visual results produced by varying the smoothness bias value.

MATERIAL ID GATHERING

Unreal Engine uses a specialized lightweight pipeline state object for the initial reflection ray tracing. It consists of a single ray generation shader, a trivial miss shader, and a common tiny closest-hit shader for all geometry in the scene. Shown in Listing 48-2, this shader aims to find the closest intersections without incurring any shading overhead.

Listing 48-2. *The material ID gathering closest-hit shader.*

```
struct FDeferredMaterialPayload
{
    float  HitT;       // Ray hit depth or -1 on miss
    uint   SortKey;    // Material ID
    uint   PixelCoord; // X in low 16 bits, Y in high 16 bits
};

[shader("closesthit")]
DeferredMaterialCHS(
    FDeferredMaterialPayload Payload,
    FDefaultAttributes Attributes)
{
    Payload.SortKey = GetHitGroupUserData(); // Material ID
    Payload.HitT    = RayTCurrent();
}
```

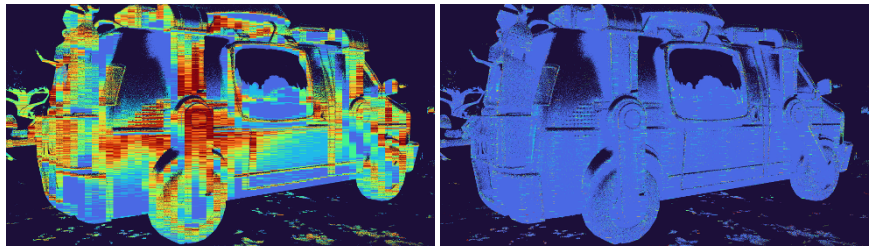
Results of the material ID gather pass are written to a deferred material payload buffer in 64×64 tile order for subsequent sorting.

RAY SORTING

Reflection ray hit points are sorted by material ID using a compute shader. Sorting is performed in blocks of 64×64 pixel screen-space tiles (4,096 total pixels). Rather than performing a full sort, rays are binned into buckets per tile. The number of buckets is the number of total pixels in a tile divided by an expected thread group size, e.g., 4,096 pixels / 32 threads = 128 buckets. There may be many more different materials in the full scene (there are approximately 500 materials in an average *Fortnite* RTPSO), but it is unlikely that a given tile will contain all of them. If there are more than 128 different materials in a tile, it is not possible to sort them into perfectly coherent groups anyway. Increasing the number of bins does not improve much beyond reducing the chance of collisions in the material ID $\rightarrow$ bucket ID mapping. In practice, we did not see any efficiency improvement from increasing the number of buckets.

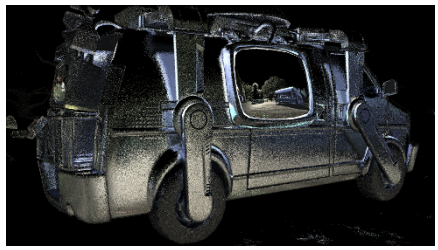
The binning is done as a single compute shader pass with one thread group per tile, using `groupshared` memory to store intermediate results. A straightforward binning algorithm is used here: (1) load elements from the deferred material buffer, (2) count the number of elements per sorting bucket using atomics, (3) build a prefix sum over the counts to compute the sorted index, and (4) write the elements back into the same deferred material buffer for material evaluation. Note that the original ray dispatch index must be preserved throughout the full pipeline and is read by hit shaders from the ray payload structure (the `DispatchRaysIndex()` intrinsic may not be used outside of the original (unsorted) ray generation shader).

As demonstrated by Figures 48-7 and 48-8, the sorting scheme reduces shader execution divergence quite effectively. Most waves in a typical *Fortnite* frame contain a single material shader when sorting is used. Although this is effective, note that the GPU performance impact is very scene dependent. For cases where most rays naturally hit the same material or the sky, there may be no performance improvement or even a small slowdown due to the sorting overhead. However, for complex scenes with many high-roughness surfaces, the speedup may be as high as $3\times$. The average performance improvement in *Fortnite* is around $1.6\times$. Sorting is used for reflections on everything except water. Reflection rays from water are highly coherent and typically hit the sky, so there is little benefit from sorting.



(a) *Material sorting disabled* **(b)** *Material sorting enabled*

Figure 48-7. *A visualization of the SIMD execution efficiency improvement from ray sorting. Dark blue areas belong to waves that did not require material shaders at all (rays that hit the sky or were culled by the roughness threshold). Brighter colors show waves that contained between 1 (light blue) and 8+ (dark red) different shaders.*



(a) *Raw reflection output*

Sorting Disabled	**4.17 ms**
Tracing	0.62 ms
Shading	3.55 ms
Sorting Enabled	**2.12 ms**
Tracing	0.62 ms
Sorting	0.15 ms
Shading	1.35 ms

(b) *GPU time measurements*

Figure 48-8. *Sorting performance comparison on the NVIDIA RTX 3090 at* 1920 × 1080 *resolution.*

MATERIAL EVALUATION

The material evaluation step loads ray parameters from a buffer that was written during the initial ray generation phase, but shortens rays to cover only a small segment around the triangle that was previously hit. Full material shaders are then invoked using `TraceRay`. We have found that despite the extra cost of tracing a second ray, this approach is significantly faster than naive `TraceRay` in a monolithic reflection pipeline.

Unreal Engine uses platform-specific APIs to launch closest-hit shaders directly, without incurring the traversal cost where possible. A viable alternative path on PC could be to use DXR callable shaders for all closest-hit shading, while using any-hit shaders for alpha mask evaluation. This comes with its own set of trade-offs, such as the need for all ray generation shaders

to explicitly pass hit parameters to callable shaders via the payload. Ultimately, the shortened ray approach was a good compromise between performance and simplicity at the time.

Avoiding any-hit shader processing as much as possible is an important performance optimization when ray tracing. Unfortunately, typical game scenes do contain alpha-masked materials that must look correct in reflections. We found that the vast majority of reflection rays in practice tend to hit either entirely opaque materials or opaque parts of alpha-masked materials, such as the solid part of a tree leaf mask texture. This makes it possible to evaluate opacity in the closest-hit shader and write the opacity status into the ray payload. All initial ray tracing can then use `RAY_FLAG_FORCE_OPAQUE`, and the ray generation shader gains the ability to decide if a "full-fat" ray needs to be traced without the `FORCE_OPAQUE` flag. This is shown in Listing 48-3.

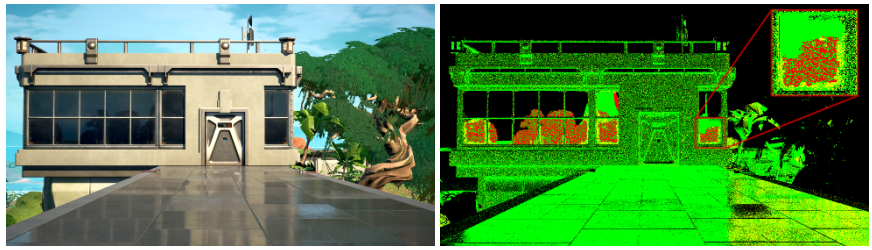
Listing 48-3. *Pseudocode for alpha-masked material ray culling in the ray generation shader.*

```
TraceRay(TLAS, RAY_FLAG_FORCE_OPAQUE, ..., Payload);
if (GBuffer.Roughness <= Threshold && Payload.IsTransparent())
{
    TraceRay(TLAS, RAY_FLAG_NONE, ..., Payload);
}
```

Fortnite uses an aggressive any-hit roughness threshold of 0.1, meaning that alpha-masked materials, such as vegetation, are rendered fully opaque in reflections on rough surfaces. Only near-perfect mirrors show the proper alpha cutouts, as shown in Figure 48-9. Though this is a quality concession, it works fairly well for *Fortnite* in practice. Performance improvement from this optimization depends on the scene, but we measured approximately 1.2× speedup on average in *Fortnite*, with some scenes approaching 2×. We found that the benefit of `FORCE_OPAQUE` *easily* pays for the cost of retracing some rays in our typical frame.

LIGHTING

Direct lighting evaluation in Unreal Engine's ray traced effects is mostly split between ray generation and miss shaders. Only emissive and indirect lighting comes from closest-hit shaders, as it may involve reading from textures or light maps. The ray generation shader contains a light loop with grid-based culling and light shape sampling. Shadows for lights in reflections are always calculated using ray tracing (instead of shadow mapping). Light irradiance is



(a) *Final image with reflections* **(b)** *Ray category visualization*

Figure 48-9. *A visualization of any-hit material evaluation. Green areas show reflection rays that hit opaque geometries or opaque parts of alpha-masked materials (as reported by the closest-hit shader). Yellow areas show where non-opaque ray tracing was skipped due to the roughness threshold. Red areas show where non-opaque rays were traced. Performance on NVIDIA RTX 3090 at* 1920 × 1080 *resolution was 1.8 ms with opaque ray optimization and 2.4 ms without.*

computed in a miss shader. If a light does not use shadows, it still goes through the common `TraceRay` path with a forced miss by setting `TMin = TMax` and `InstanceInclusionMask = 0`. This is similar to launching a callable shader, but avoids an extra transition in and out of the ray generation shader. Using miss shaders in this way slims down the ray generation shader code and results in better occupancy, leading to a performance increase on all of our target platforms.

This design also improves SIMD efficiency during lighting, as rays within one wave may hit different materials. Execution diverges during material evaluation, but re-converges for lighting. Material sorting does not fully solve the divergence problem as it's not possible to always perfectly fill waves, leaving some waves only partially utilized.

Keeping all lighting calculations in the ray generation shaders allows for smaller closest-hit shaders. This is beneficial in many ways, from iteration speed to code modularity and game patch sizes. Unreal Engine uses a single common set of material hit shaders for all ray tracing effects and a single main material ray payload structure, shown in Listing 48-4. There is a trade-off akin to forward versus deferred shading in raster graphics pipelines, where the G-buffer provides an opaque interface between different rendering stages and allows decoupled/hot-swappable algorithms. However, this comes at the cost of a large G-buffer memory footprint or a larger ray payload structure.

Listing 48-4. *The standard UE4 ray tracing material ray payload.*

```
struct FPackedMaterialClosestHitPayload
{
    float HitT                                          // 4  bytes
    uint PackedRayCone;                                 // 4  bytes
    float MipBias;                                      // 4  bytes
    uint RadianceAndNormal[3];                          // 12 bytes
    uint BaseColorAndOpacity[2];                        // 8  bytes
    uint MetallicAndSpecularAndRoughness;               // 4  bytes
    uint IorAndShadingModelIDAndBlendingModeAndFlags;   // 4  bytes
    uint PackedIndirectIrradiance[2];                   // 8  bytes
    uint PackedCustomData;                              // 4  bytes
    uint WorldTangentAndAnisotropy[2];                  // 8  bytes
    uint PackedPixelCoord;                              // 4  bytes
}; // 64 bytes total
```

LIGHT SOURCE CULLING

In addition to optimizing the material evaluation costs, we needed to balance the cost of illuminating reflected surfaces. *Fortnite*'s expansive world creates scenarios where a large number of lights may potentially impact a surface. As surfaces often come close to being affected by up to 256 lights (the maximum supported in reflections), we required a strategy to select only the lights that meaningfully affect a surface. Our chosen approach uses a world-aligned, camera-centered 3D grid to perform light culling.

Fortnite's large world requires a trade-off with cell sizes: large cells hurt the efficiency of culling, but small cells are not practical due to the required coverage area. A compromise was made with cells increasing in size exponentially based on the distance to the camera. To allow the cells to fit together in a grid, scaling is applied independently for each axis. This produces modest 8 meter^3 ($2 \times 2 \times 2$) cells near the camera, while still having discrete cells 100 meters from the camera. A further tweak, which simplifies the mathematics, keeps the first two layers of cells the same size. Figure 48-10 shows the layout of the grid in two dimensions, and Listing 48-5 shows the HLSL shader code used to compute a cell address for an arbitrary position in world space.

Shown in Figure 48-11, the final representation of the light culling data is implemented as a three-level structure on the GPU. The grid is the topmost structure, with 128 bits stored per grid cell that encodes either a list of up to 11 indices of 10 bits each or a count and offset into an auxiliary buffer. This auxiliary buffer holds indices for cells that contain too many lights for the

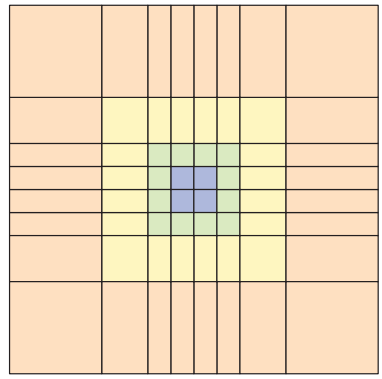


Figure 48-10. *A 2D slice of the light grid showing the four closest rings of cells.*

Listing 48-5. *HLSL code for light grid cell addressing.*

```
int3 ComputeCell(float3 WorldPosition)
{
    float3 Position = WorldPos - View.WorldViewOrigin;
    Position /= CellScale;

    // Use symmetry about the viewer.
    float3 Region = sign(Position);
    Position = abs(Position);

    // Logarithmic steps with the closest cells being 2x2x2 scale units
    Position = max(Position, 2.0f);
    Position = min(log2(Position) - 1.0f, (CellCount/2 - 1));

    Position = floor(Position);
    Position += 0.5f;             // Move the edge to the center.
    Position *= Region;           // Map it back to quadrants.

    // Remap [-CellCount/2, CellCount/2] to [0, CellCount].
    Position += (CellCount / 2.0f);

    // Clamp to within the volume.
    Position = min(Position, (CellCount - 0.5f));
    Position = max(Position, 0.0f);

    return int3(Position);
}
```

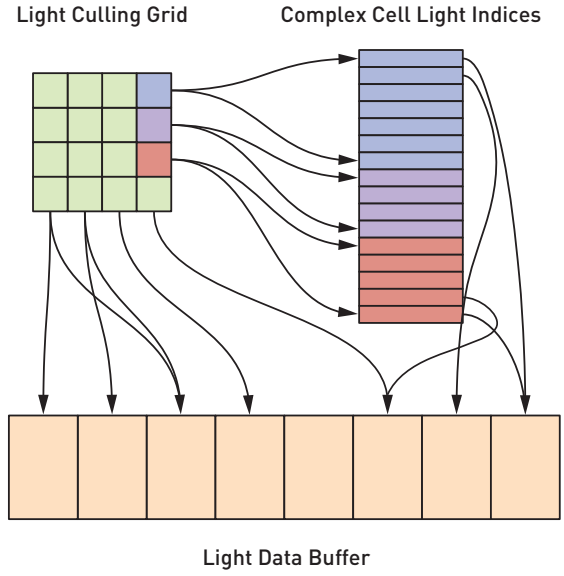


Figure 48-11. *Grid cells either address the light buffer directly or offset into an array of light indices for complex cells.*

compact format. The lowest level is a structured buffer of light data parameters to which the indices refer. Listing 48-6 shows how indices are retrieved for a grid cell. Both the grid structure and the auxiliary buffer are generated each frame by a compute shader culling lights against the grid.

48.4.2 GLOBAL ILLUMINATION

Global illumination was not an initial requirement for ray tracing in *Fortnite*. Originally released as an experimental algorithm in Unreal Engine 4.22, brute-force global illumination was not practical for uses that demanded real-time performance. Instead, the brute-force algorithm was only viable for interactive and cinematic frame rates. The original algorithm was reformulated into the "final gather" algorithm in Unreal Engine 4.24. Ongoing development, strict lighting constraints, and substantial qualitative improvements from denoising and upscaling led to adoption of the final gather approach as a potential real-time global illumination solution for *Fortnite*. In this section, we outline the two algorithms and discuss the technological improvements and optimizations that led to achieving real-time performance. Figure 48-12 shows the visual result achieved in-game.

Listing 48-6. *HLSL code to retrieve a light index for a grid cell.*

```
int GetLightIndex(int3 Cell, int LightNum)
{
    int LightIndex = -1; // Initialized to invalid
    const uint4 LightCellData = LightCullingVolume[Cell];

    // Whether the light data is inlined in the cell
    const bool bPacked = (LightCellData.x & (1 << 31)) > 0;

    const uint LightCount = bPacked ? (LightCellData.w >> 20) & 0x3ff
                            : LightCellData.x;

    if (bPacked)
    {
        // Packed lights store 3 lights per 32-bit quantity.
        uint Shift = (LightNum % 3) * 10;
        uint PackedLightIndices = LightCellData[LightNum / 3];
        uint UnpackedLightIndex = (PackedLightIndices >> Shift) & 0x3ff;

        if (LightNum < LightCount)
        {
            LightIndex = UnpackedLightIndex;
        }
    }
    else
    {
        // Non-packed lights use an external buffer
        // with the offset in the cell data.
        if (LightNum < LightCount)
        {
            LightIndex = LightIndices[LightCellData.y + LightNum];
        }
    }
    return LightIndex;
}
```

BRUTE FORCE

The experimental brute-force algorithm uses Monte Carlo integration to solve the diffuse interreflection component of the rendering equation [8]. As with other ray tracing passes, the global illumination pass begins at the G-buffer, where diffuse rays are generated in accordance to the world-space normal of the rasterized depth-buffer's position. In this manner, the brute-force algorithm behaves very similarly to that of an ambient occlusion algorithm. However, instead of casting visibility rays to tabulate sky occlusion, the global illumination algorithm casts more expensive rays, evaluating secondary surface material information by invoking the closest-hit shader.



Figure 48-12. *Fortnite's Risky Reels, before and after applying ray traced global illumination. Note the bounce lighting from the grass to the grill.*

Along with expensive evaluation of the closest-hit shader, the algorithm must also apply direct lighting to secondary surfaces. Instead of applying the traditional light loop, the global illumination algorithm uses next event estimation. *Next event estimation* (NEE) is a stochastic process that chooses a candidate light with some probability. The first process, known as light selection, decides which light to sample, according to some selection probability. The second process, similar to traditional light sampling, samples an outgoing direction to the light source. The NEE process constructs a shadow ray to test the visibility of the shading point in relation to the selected light. If the visibility ray successfully connects to the light source, diffuse lighting evaluation is recorded.

NEE generates a smaller per-ray cost than the traditional lighting loop, as it only evaluates a subset of the total number of candidate lights. Though the NEE is typically considered to select one light source per invocation, another secondary stochastic process may be invoked to draw multiple NEE samples. Drawing multiple samples has the effect of lowering per-ray variance while also amortizing the cost of casting an expensive material evaluation ray. We find that drawing two NEE samples per material evaluation ray works well in practice. See Listing 48-7 for an abbreviated code example.

Listing 48-7. *Next event estimation.*

```
float3 CalcNextEventEstimation(
    float3 ShadingPoint,
    inout FPayload Payload,
    inout FRandomSampleGenerator RNG,
    uint SampleCount
)
{
    float3 ExitantRadiance = 0;
    for (uint NeeSample = 0; NeeSample < SampleCount; ++NeeSample)
    {
        uint LightIndex;
        float SelectionPdf;
        SelectLight(RNG, LightIndex, SelectionPdf);

        float3 Direction;
        float Distance;
        float SamplePdf;
        SampleLight(LightIndex, RNG, Direction, Distance, SamplePdf);

        RayDesc Ray = CreateRay(ShadingPoint, Direction, Distance);
        bool bIsHit = TraceVisibilityRay(TLAS, Ray);
        if ( !bIsHit )
        {
            float3 Radiance = CalcDiffuseLighting(LightIndex, Ray, Payload);
            float3 Pdf = SelectionPdf * SamplePdf;
            ExitantRadiance += Radiance / Pdf;
        }
    }
    ExitantRadiance /= SampleCount;

    return ExitantRadiance;
}
```

Depending on the maximum number of bounces allowed by the artist, the brute-force algorithm may fire another diffuse ray from the secondary surface and repeat the process to extend the path chain. Subsequent bounces are susceptible to early termination and are governed by a Russian roulette process. Our cinematics department prefers to use two bounces of global illumination as their default configuration.

Global illumination computed in this manner matches closely with our path tracing integrator. However, also like our path tracing integrator, this process requires a high number of samples to converge. A strong denoising kernel helps to generate a smooth final result, but we find that between 16 and 64 samples per pixel are still necessary for sufficient quality, depending on the lighting conditions and overall environment. This is problematic, because casting multiple material evaluation rays per frame quickly pushes the



Figure 48-13. *The brute-force global illumination technique in our development test environment, rendered with two samples per pixel at* `ScreenResolution = 50`*. Note the yellow color bleeding from the nearby wall onto the bush.*

algorithm beyond interactive rates and is only applicable for cinematic burnouts. Figure 48-13 shows the application of the brute-force global illumination to our *Fortnite* development level.

FINAL GATHER

In an effort to keep the nice properties of the brute-force integrator, the algorithm was amended in September 2019 to render diffuse interreflection for our Archviz Interior Rendering sample [4] at a cinematic frame rate of 24 Hz.

The key insight to accelerate the brute-force algorithm involves transforming the costly material evaluation rays into relatively inexpensive visibility rays. To achieve this, we time-slice the material evaluation rays over consecutive frames. We invoke the brute-force integrator, but only at one sample per pixel, and use previous frame simulation data to accumulate as though we actually fired the desired number of samples per pixel. Accumulating previous frame data requires sample reprojection and may cause severe ghosting artifacts, especially when the previous frames' simulation data is no longer valid. To help reconcile discrepancies associated with accumulating previous frame data, we choose to employ primary path reconnection [6] to

reuse previously traced paths. Previous path data is cached in an intermediary structure that we call *gather points*. Each gather point encodes the position of the secondary surface, along with the recorded irradiance and path creation probability density function (PDF). The world position of the pixel is also cached and is used to test against reprojection criteria for reuse in subsequent frames. The gather point buffer is interpreted as a circular buffer, where the buffer length is governed by the number of samples per pixel of the algorithm. In a manner similar to Bekaert et al. [2], we fire secondary visibility rays to test for successful path reconnection. Doing so correctly requires carrying both the exitant radiance and the probability density of the gather point creation from the active shading point in the previous simulation. Like next event estimation, a successful path reconnection event records the diffuse lighting at the gather point.

A wave of diffuse rays is dispatched as an individual pass each frame. This pass has similar execution flow to the brute-force algorithm, but records lighting data to a secondary *gather point buffer*. The gather point buffer records the secondary surface position and diffuse exitant radiance from stochastic light evaluation, as well as the probability density of generating the gather point from the simulation. The original creation point is also supplied so that we can reject gather points that do not meet sufficient criteria for reuse in the current frame. From this data, we can cast path reconnection events to these points and incorporate the cached lighting evaluation. The gather points pass is accelerated using the same sorted-deferred material evaluation pipeline used with ray traced reflections. (See Listing 48-8.)

Once the gather points are created, the final gather pass is executed. The final gather pass loops through all gather points associated with the current pixel and reprojects them to the active frame. Gather points that successfully reproject are candidates for path reconnection. A visibility ray is fired to potentially connect the world position of the shading point to the world position of the gather point. If a path reconnection attempt is successful, the shading point records diffuse lighting from the gather point. We found that we still needed approximately 16 path reconnection events to have good qualitative results. Figure 48-14 and Table 48-1 present visual and runtime comparisons, respectively, of the two global illumination methods.

Listing 48-8. *Individual gather samples are cached in a structured buffer of gather points where the dimensions are governed by the pass resolution and the samples per pixel. To conserve space, the irradiance field of a gather point encapsulates both the irradiance and creation PDF of the gather sample.*

```
struct FGatherSample
{
    float3 CreationPoint;
    float3 Position;
    float3 Irradiance;
    float Pdf;
};

struct FGatherPoint
{
    float3 CreationPoint;
    float3 Position;
    uint2 Irradiance;
};

uint2 PackIrradiance(FGatherSample GatherSample)
{
    float3 Irradiance = ClampToHalfFloatRange(GatherSample.Irradiance);
    float Pdf = GatherSample.Pdf;

    uint2 Packed = (uint2)0;
    Packed.x = f32tof16(Irradiance.x) | (f32tof16(Irradiance.y) << 16);
    Packed.y = f32tof16(Irradiance.z) | (f32tof16(Pdf) << 16);
    return Packed;
}

FGatherPoint CreateGatherPoint(FGatherSample GatherSample)
{
    FGatherPoint GatherPoint;
    GatherPoint.CreationPoint = GatherSample.CreationPoint;
    GatherPoint.Position = GatherSample.Position;
    GatherPoint.Irradiance = PackIrradiance(GatherSample);
    return GatherPoint;
}
```

The final gather algorithm is limited to one bounce of diffuse interreflection. The technique could be extended to multiple bounces, by allowing stochastic gather point creation at a given event, but for simplicity we opted for one bounce. Because artificially limiting the bounce count has the impact of both lower runtime cost and lower variance, we felt this was an appropriate compromise given the general expense of the technique. Unfortunately, reprojection and path reconnection failures can result in slower convergence compared to the brute-force method. This is possible when experiencing significant camera or object motion.

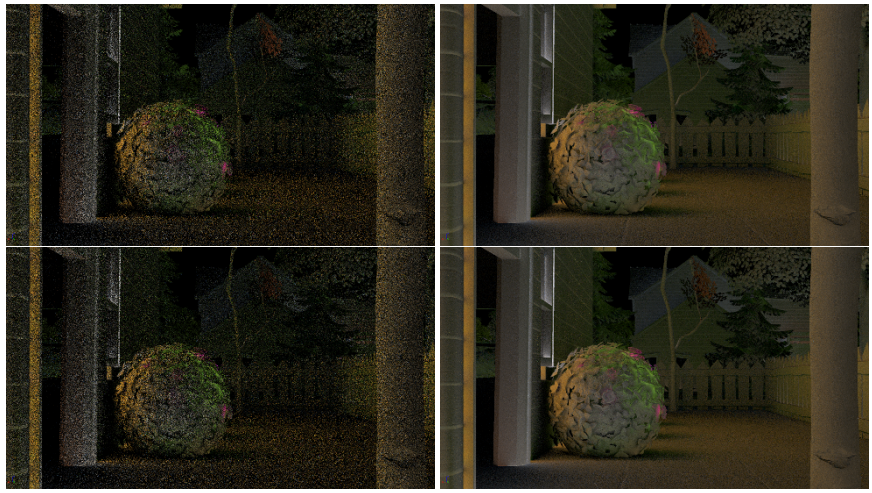


Figure 48-14. *Results from the two global illumination algorithms. Top: the brute-force method. Bottom: the final gather method. Each result is rendered at* `ScreenPercentage = 100` *for clarity and shown at one sample per pixel (left) and 16 samples per pixel (right). The images have been brightened for visualization purposes.*

SPP	Brute Force (ms)	Final Gather (ms)
1	19.78	11.63
2	46.95	13.39
4	121.33	13.49
8	259.48	15.86
16	556.31	20.15

Table 48-1. *GPU times for the global illumination passes presented in Figure 48-14 at* `ScreenPercentage = 50` *using an NVIDIA RTX 2080 Ti.*

FEASIBILITY FOR FORTNITE

Ray traced global illumination development was suspended shortly after our UE 4.24 release. With Unreal Engine 5 technology entering full development, it no longer made sense to extend an algorithm that would ultimately compete against newer initiatives.

Technologies such as NVIDIA Deep Learning Super Sampling (DLSS) [11, 3, 9] started new discussions, however. Early experiments with our test *Fortnite* level revealed that, with DLSS in performance mode, we could run our entire suite of ray tracing shaders at roughly 50 frames per second! Ray traced

shadow, ambient occlusion, reflection, sky light, and global illumination algorithms ran close to our intended budget for release. This proved to be very exciting initially, but the production map was still much more complex. For global illumination, in particular, the primitive stochastic light selection method could not cope with *Fortnite*'s large active light count. Even with necessary improvements to light selection, significant sample noise (mostly with interior lighting) made the final gather algorithm a difficult prospect for adoption.

As happens in production, other feature targets changed over the course of development, too. One of the most notable decisions was the omission of ray traced shadows for all lights except the sun (directional light). We concluded that if ray traced shadows were only included for the sun, we could apply similar exclusionary rules for global illumination as well. Of course, this also restricted bounce lighting to exterior environments, but if we limited global illumination to the sun, we could eliminate our algorithmic inefficiency with regard to light selection. Other potential sampling issues, such as illumination from large area lights, also went away. By adjusting the next event estimation samples to one, we avoided the cost of firing a second shadow ray and gained additional savings. With the feature set significantly culled, employing ray traced global illumination actually seemed feasible.

IMPROVEMENTS

Despite culling a large swath of algorithmic requirements, deploying the final gather algorithm for *Fortnite* was still challenging, due to remaining performance and sampling issues.

It became clear that we needed to operate at a coarser resolution to remain in budget. We expected that operation at half resolution or smaller would be required for the project. Unfortunately, we found that operating at smaller resolutions typically required more reconnection events to help mitigate noise. Doing so was not compatible with our temporal strategy; however, the likelihood of successful reconnection events diminished as our temporal lag increased. Overall dependence on temporal history proved difficult for a fast-moving game like *Fortnite*.

We started experimenting with extended strategies for gather point reprojection and path reconnection. Our simple, world-based reprojection of gather point data was improved with time-varying camera projection to improve stability for fast-moving camera motion. Though the first

implementation of the final gather algorithm exploits temporal path reprojection of gather points, we expected that both spatial and temporal reuse would be necessary to increase our effective samples per pixel. Previous experiments in this area had failed, and we discovered, as was also previously presented by Bekaert et al. [2], that using a regular neighborhood reconstruction kernel exhibited strongly objectionable structured noise, despite the observed error reduction in the final result.

Further experimentation in this area was halted as we also had to address our general denoising problem. Our previous global illumination denoiser operates well at full resolution, but quickly degrades when rendering at lower resolutions. Thankfully, NVIDIA presented us with an implementation of spatiotemporal variance-guided filtering (SVGF) [13]. SVGF proved to be a game-changer for the global illumination algorithm. We found that SVGF tolerated downsampled, noisy images with better results than our integrated denoiser. SVGF readily accepted the structured noise present with the newer spatial path reconnection strategy and generated very pleasing results while increasing the overall effective samples per pixel. Unfortunately, our implementation of SVGF exhibited bleeding artifacts with high-albedo surfaces when attempting to upscale to the required resolution (Figure 48-15). Though the occurrence was not frequent, it was prevalent in the Sweaty Sands



Figure 48-15. *Fortnite's Misty Meadows, before and after applying ray traced global illumination. Note the red color bleeding from the rooftops onto the buildings.*



(a) *No spatial gather point reconnection* **(b)** *Spatial gather point reconnection*

Figure 48-16. *Fortnite's Sweaty Sands, before and after applying ray traced global illumination. Sharing neighborhood samples creates strong structured artifacts, but SVGF is still able to reconstruct a smooth result while also dampening temporal artifacts.*

point of interest (see Figure 48-16) and needed to be addressed. Facing tough deadlines, we opted to implement an upsampling prepass so that G-buffer associations would not confuse the filter. We intend to address this exorbitant cost in the future, should we employ SVGF on another project.

We present a final breakdown of iterative improvements to the final gather algorithm in Table 48-2 and a final cost breakdown per pass in Table 48-3. Limiting global illumination to the directional light shows a significant cost savings when avoiding light selection. DLSS allows the method to work at a fractional scale, applying another significant speedup. A moderate speed gain is achieved by limiting lighting to one next event estimation sample. Costs are reintroduced to maintain temporal stability, when applying our spatial reconnection strategy with SVGF. SVGF offers pleasing results at both half and quarter resolution dimensions, which drives our *high* and *low* quality settings.

48.4.3 CPU OPTIMIZATIONS

GPU BUFFER MANAGEMENT

While profiling the CPU cost when ray tracing is enabled, we quickly noticed that our D3D12 data buffer management code needed to be improved. A lot of

Improvement	Time (ms)
Final gather baseline	10.25
Directional light only	6.14
1 NEE sample	5.59
DLSS mode: Performance	2.52
Spatial reconnection	2.78
SVGF denoiser	4.42
Screen percentage 25%	3.22

Table 48-2. *Incremental improvements applied to Figure 48-15. The last two entries correspond to the current global illumination user settings, high and low, respectively. GPU times reported at 1080p resolution on an NVIDIA RTX 2080 Ti*

Pass	Low Quality (ms)	High Quality (ms)
Gather point tracing	0.18	0.23
Gather point sorting	0.02	0.03
Gather point shading	0.49	0.92
Final gather tracing	0.49	1.18
SVGF	2.03	2.03

Table 48-3. *Per-pass execution costs associated with rendering Figure 48-15. SVGF costs are constant due to our need to run an upsampling prepass to the required DLSS resolution. GPU times reported at 1080p resolution on an NVIDIA RTX 2080 Ti.*

time in *Fortnite* was spent creating and destroying acceleration structure data buffers during gameplay due to mesh data streaming.

An easy optimization was to sub-allocate all these resources from dedicated heaps instead of using individual committed or placed resources. This is possible because acceleration structure buffers must be kept in the D3D12 `RAYTRACING_ACCELERATION_STRUCTURE` state, while scratch buffers are kept in the `UNORDERED_ACCESS` state. This means that there are no state transitions and no per-buffer state tracking required. This adjustment also saves a lot of memory due to smaller alignment overhead (placed resources in D3D12 require a 64K alignment, but most buffers are a lot smaller).

We had to make sure that committed resources are never created during gameplay, as this can introduce huge CPU spikes (sometimes 100+ ms, in our measurements). The buffer pooling scheme in UE4's D3D12 backend was adjusted to support the large allocations needed for top- and bottom-level acceleration structure data (the maximum allocation size was increased).

Readback buffers, used to get the information for compaction, are pooled and sub-allocated as placed resources from a dedicated heap.

Another problem is that almost all static bottom-level acceleration structure (BLAS) buffers were temporarily created at full size and then compacted. Compaction requires readback of the final BLAS size and a copy into the new compacted acceleration structure buffer. This can cause a lot of fragmentation and memory waste. We did not implement pool defragmentation for *Fortnite* due to time constraints, but this was done later for Unreal Engine 5.

DYNAMIC RAY TRACING GEOMETRIES

Another substantial CPU bottleneck we faced was in collecting and updating all the dynamic meshes in the scene before BLAS data is updated. A compute shader is run for each mesh to generate the dynamic vertex data for that frame. This temporary vertex data is then used to update/refit the BLAS. Potentially hundreds of dispatches and build operations can be kicked off in a single frame. Each dispatch might use a different compute shader and output vertex buffer, causing a lot of CPU overhead when generating the command lists. This performance overhead comes from switching the shaders/state and from binding different shader parameters.

We first optimized this process by sorting all dynamic geometry update requests by shader, but it was not quite enough. Additional overhead comes from switching the buffers for each mesh and performing the internal resource state transitions. Most dynamic meshes, such as characters or deformable objects, need to be updated every frame so the updated vertex data does not have to be persistently stored. This allowed us to use transient per-frame buffers with simple linear sub-allocation within them per mesh, minimizing the cost for state tracking. Each compute shader dispatch writes to a different part of the buffer; therefore, unordered access view (UAV) barriers are not necessary between dispatches. Finally, we parallelized the generation of the dynamic geometry update command list with the BLAS update command list to hide the surprisingly high `BuildRaytracingAccelerationStructure` CPU costs.

BUILDING THE SHADER BINDING TABLES

The largest CPU cost (by far) is building the ray tracing shader binding table (SBT) for each scene and ray tracing pipeline state object. The Unreal Engine

does not use persistent shader resource descriptor tables, so all resource bindings have to be manually collected and copied into a single shared descriptor heap every frame.

To reduce the cost of copying all the descriptors for all the meshes, we introduced several levels of caching. The biggest win came from deduplicating SBT entries with the same shader and resources, such as the same material being applied to different meshes or multiple instances of the same mesh. Unreal Engine groups shader resource bindings into high-level tables ("Uniform Buffers"), and a typical shader references three or four of them (view, vertex factory, material, etc.). Each uniform buffer in turn may contain references to textures, buffers, samplers, etc. We can cache SBT records by simply looking at the high-level uniform buffers without inspecting their contents.

A lower-level cache was added to deduplicate the actual resource descriptor data in descriptor heaps. This was mostly needed to deduplicate the sampler descriptors, as a single D3D12 sampler heap can only have 2048 entries, and only one sampler heap can be bound at a time. We found that the cost of the D3D12 `CopyDescriptors` call is roughly the same as (or larger than) the cost of descriptor hashing and hash table lookup/insert.

Finally, we parallelized the SBT record building, but found that improvements from adding worker threads diminished very quickly. As we still wanted to use descriptor deduplication, each worker thread needed to use its own local descriptor cache to avoid synchronization overheads. Global descriptor heap space is then allocated by each worker in chunks, using atomics. This reduces the cache efficiency and increases the total number of used descriptor heap slots. We found that four or five worker threads is the sweet spot for parallel SBT generation.

GEOMETRY CULLING

A simple but effective technique used to speed up both CPU and GPU render times was to skip instances entirely if they were not relevant to the current frame. When ray tracing, every object in the scene can affect what is seen by the camera. However, in real life the contribution of many objects can be negligible. Initially, we experimented with culling geometries behind the camera that were placed further than a certain distance threshold. However, this solution was discarded because it was producing popping artifacts when objects with large coverage where rejected in a frame and accepted in the

next one. The solution was to change the culling criteria to also take into account the projected area of the bounding sphere for the instance, and to discard it only if it was small enough. This simple change removed the popping while greatly improving speed. On average, we observed gains of 2–3 ms per frame after making this change.

DLSS

The integration of NVIDIA's DLSS technology was instrumental in improving performance. DLSS made it possible to enable ray traced global illumination and run the game with ray tracing enabled at higher resolutions. The engine changes made to integrate DLSS are available now in the public UE4 codebase [5] and the DLSS plugin for UE4 is now available on the Unreal Engine Marketplace [12].

48.5 FORTNITE CINEMATICS

Though it took some time until ray tracing was used in the game, the *Fortnite* team adopted ray tracing from the very early days for other purposes such as cinematic trailers and marketing content. Moving to a ray tracing pipeline allowed artists to reduce iteration times and increase the overall quality achieved. It also helped lighting artists with a background in offline rendering to speed up their initial learning of Unreal Engine because they were already familiar with similar rendering tools. An interesting aspect of this work is that, though the use cases are different and the quality requirements are higher than in the game, these cinematographic pieces are made using the same technology and assets used in the game.

48.6 CONCLUSION

Ray Tracing was shipped in *Fortnite* Season 15 (September 2020) and achieved more ambitious goals than what we initially set out to accomplish. Though the project was challenging on many levels, it ended up being a success. Not only does the game look better when ray tracing is enabled, but all the improvements are now part of the UE4 public codebase [5].

This initiative is not yet finished. There are many open problems in real-time ray tracing that need more work, including scenarios with a high number of dynamic geometries that must be updated every frame, complex light transport that requires multi-bounce scattering, and scenes with large amounts of dynamic lights. These problems are difficult and will require

multi-year efforts from the computer graphics community. The engineering team at Epic Games will continue improving the technologies developed for this project, as well as additional novel methods that we have on our road map, with the goal of making ray tracing a feasible solution for any kind of game or real-time graphics application.

ACKNOWLEDGMENTS

We would like to thank the rendering team at Epic Games for their help and feedback: Yujiang Wang, Guillaume Abadie, Chris Bunner, Michal Valiant, Marcus Wassmer, Kim Libreri, and all the great engineers that have contributed to make ray tracing in Unreal Engine.

Many thanks also to the *Fortnite* art ninjas: Jordan Walker, Mike Mulholland, Andrew Harris, Juan Collado, Kirsten Todd, Paul Mader, Alex Gonzalez, Luke Tannenbaum, Maddie Duque, Tim Elek, and many others.

Also, this endeavor would not have been possible without the amazing support the production and QA teams gave us: Scott James, Paul Oakley, Shak Khavarian, Kirstin Riddle, Katie McGovern, Petra Pintar, Ari Patrick, Brandon Grable, Stan Hormell, Will Fissler, Zachary Wilson, etc.

The authors would also like to thank the book editors for their very helpful suggestions and insights.

REFERENCES

[1] Aalto, T. Bringing ray tracing into Remedy's Northlight engine. https://www.youtube.com/watch?v=ERlcRbRoJF0, Nov. 22, 2018. Accessed May 05, 2021.

[2] Bekaert, P., Sbert, M., and Halton, J. Accelerating path tracing by re-using paths. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 125–134, 2002.

[3] Edelsten, A., Jukarainen, P., and Patney, A. Truly next-gen: Adding deep learning to games and graphics. Presentation at Game Developers Conference, https://www.gdcvault.com/browse/gdc-19/play/1026184, 2019.

[4] Epic Games. New Archviz interior rendering sample project now available! https://www.unrealengine.com/en-US/blog/new-archviz-interior-rendering-sample-project-now-available, Dec. 19, 2019.

[5] Epic Games. Unreal Engine source code repository. https://www.unrealengine.com/en-US/ue4-on-github, Jan. 1, 2021.

[6] Fascione, L., Hanika, J., Heckenberg, D., Kulla, C., Droske, M., and Schwarzhaupt, J. Path tracing in production: Part 1: Modern path tracing. In *ACM SIGGRAPH 2019 Courses*, 19:1–19:113, 2019. DOI: 10.1145/3305366.3328079.

[7] Heitz, E. Sampling the GGX distribution of visible normals. *Journal of Computer Graphics Techniques*, 7(4):1–13, 2018. http://jcgt.org/published/0007/04/01/.

[8] Kajiya, J. T. The rendering equation. 20(4), 1986.

[9] Liu, E. DLSS 2.0: Image reconstruction for real-time rendering with deep learning. Presentation at Game Developers Conference, https://www.gdcvault.com/play/1026697/DLSS-Image-Reconstruction-for-Real, 2020.

[10] Liu, E., Llamas, I., Cañada, J., and Kelly, P. Cinematic rendering in UE4 with real-time ray tracing and denoising. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 19, pages 289–319. Apress, 2019.

[11] NVIDIA. Deep Learning Super Sampling (DLSS). https://www.nvidia.com/en-us/geforce/technologies/dlss/, 2019. Accessed December 19, 2019.

[12] NVIDIA. DLSS plugin for Unreal Engine 4. *Unreal Engine Marketplace*, https://www.unrealengine.com/marketplace/en-US/product/nvidia-dlss, Feb. 10, 2021.

[13] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, 2:1–2:12, 2017. DOI: 10.1145/3105762.3105770.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 49

REBLUR: A HIERARCHICAL RECURRENT DENOISER

Dmitry Zhdan
NVIDIA

ABSTRACT

This chapter presents ReBLUR: a hierarchical recurrent denoiser, currently deployed in recent RTX game titles, including *Cyberpunk 2077* and *Watch Dogs: Legion*. By aggressively reusing information both spatially and temporally, the denoiser produces noise-free images from only a few samples per pixel. In this chapter, we describe the internal structure of the denoiser in detail, and we hope that the basic blocks of this architecture can be useful to improve future denoisers.

49.1 INTRODUCTION

Ray tracing is increasingly popular in real-time rendering applications. To keep frame rate high, a low number of rays per pixel (rpp) is used. For example, with one ray per pixel in checkerboard mode, this means just 0.5 rpp. With a low number of ray counts per pixel, the generated image often contains significant noise. To remove this noise, a *denoiser* pass is commonly applied, which reconstructs a high-quality final image from the noisy signal, often with help of G-buffer feature guides. In a real-time ray tracing setting, a denoiser must support a signal with low rpp, produce minimal (or acceptable) temporal lag, run in a few milliseconds even at large resolutions, and converge to something very close to the ground truth.

To meet these goals, temporal accumulation is an essential component, combined with accurate specular reprojection (specular motion does not follow surface motion). Furthermore, to avoid over-blurring or under-blurring, the spatial filter weights must be carefully selected. Care must also be taken to design a fallback if history gets rejected (the input signal needs to be heavily processed to avoid falling back to the pure 1 rpp noisy input). Due to the temporal component of the denoiser, temporal lag is introduced, which can be visible when a scene rapidly changes (e.g., a change

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_49

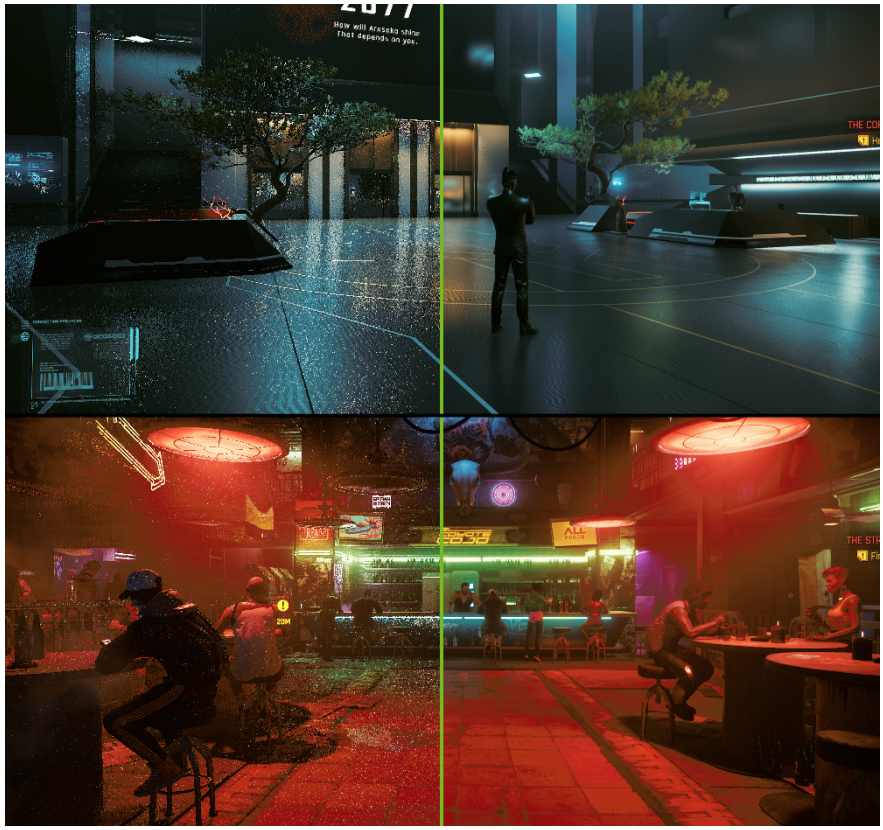


Figure 49-1. *Cyberpunk 2077 with ray tracing. Left: diffuse (1 rpp) and specular (0.5 rpp) noisy inputs. Right: denoised result from ReBLUR.*

in lighting or in scene geometry). To combat temporal lag, an *anti-lag* solution is needed, which can explicitly control the amount of accumulated frames and adjust accumulation speed if such changes are detected.

NVIDIA has developed two spatiotemporal denoising algorithms solving these problems: ReBLUR and ReLAX. ReBLUR is based on recurrent blurring, which applies cross-bilateral spatial filtering to the same texture recursively. ReLAX is an advanced version of spatiotemporal variance guided filtering (SVGF), its main feature is clamping to the fast history. It has been designed to work with signal generated by the NVIDIA RTX Direct Illumination algorithm (RTXDI). These two denoising solutions are different, but serve the same goal: delivering the best denoising in games. Both solutions are part of the NVIDIA Real-Time Denoisers library (NRD) [3].

This chapter is about ReBLUR. In Figure 49-1, we show ReBLUR in *Cyberpunk 2077*. In the following sections, we will outline the main principles of the approach.

49.2 DEFINITIONS AND ACRONYMS

`uv`	Screen-space coordinates, normalized to the range [0, 1].
$\mathbf{n}$	Surface normal vector in world space.
`roughness`	Linear roughness.
X	World-space position in camera relative space.
X_{prev}	World-space position in the previous frame.
X_v	View-space position.
$\mathbf{v}$	View vector pointing out from the surface.
$\mathbf{r}$	Reflection vector = `reflect`($-\mathbf{v}$, $\mathbf{n}$).
`viewZ`	Linearized view depth.
A	Number of accumulated frames.
$\mathbf{n} \cdot \mathbf{v}$	Dot product between normal and view vectors, `NoV`.
$\mathbf{d}_{GGX}$	Specular dominant direction.

49.3 THE PRINCIPLE

ReBLUR is based on recurrent spatiotemporal filtering [8], where the denoised result from the previous frame is used as the history buffer in the current frame. It means that the result of spatial passes goes back into the temporal feedback loop to be accumulated with the input signal on the next iteration. This approach allows us to redistribute spatial filtering in time and use a lower number of samples per spatial pass. ReBLUR is a hierarchical denoiser, i.e., it starts with very blurry results and rapidly converges over time. The spatial filters are applied in world space, using stochastically rotated Poisson disk sampling. The spatial filters adapt to the number of accumulated frames and become less aggressive over time, until a disocclusion is found or the internal anti-lag invalidates the history. In Figure 49-2 we show ReBLUR applied to a highly reflective version of the Lumberyard Bistro scene.



Figure 49-2. *Lumberyard Bistro with ray traced cobalt material. Left: noisy input (0.5 rpp). Right: denoised result with ReBLUR.*

49.4 INPUTS

ReBLUR has been designed to be used in modern games and requires the following inputs for each pixel (see Listing 49-1):

n	World-space surface normal.
`viewZ`	Linearized view depth (to reconstruct primary hit position).
`roughness`	Linear roughness.
`motion vector`	World-space surface motion: $X_{prev} = X + $ `motion vector`.
$\mathbf{c}_{diffuse}$	Noisy diffuse signal.
$\mathbf{c}_{specular}$	Noisy specular signal.
`hitT`	Hit distance, which controls the filter radius in spatial passes and for specular reprojection.

It's worth noting that world-space position reconstruction in ReBLUR is not compatible with the Primary Surface Replacement method, used for rendering perfect reflections and refractions in path traced games [4], because in this case the real first primary hit is replaced with a potentially unlimited number of bounces through true mirrors, which can't be reconstructed using a `viewZ` texture. Explicit per-pixel world-space position is needed to solve this issue.

Listing 49-1. *View-space position reconstruction from linearized view depth.*

```
float3 ReconstructViewPosition( float2 uv, float4 cameraFrustum, float viewZ
    , float isOrtho = 0.0 )
{
    float3 p;
    p.xy = uv * cameraFrustum.zw + cameraFrustum.xy;
    //isOrtho = { 0 - perspective, -1 - right handed ortho, 1 - left handed
        ortho }
    p.xy *= viewZ * ( 1.0 - abs( isOrtho ) ) + isOrtho;
    p.z = viewZ;

    return p;
}
```

49.5 PIPELINE OVERVIEW

Diffuse and specular denoisers share the same pipeline, which is the sequence of the passes shown in Figure 49-3.

49.5.1 PRE-BLUR

All blur passes are similar, but the main idea of the *pre-blur* pass is to blur out outliers. ReBLUR doesn't suppress them, but tries to redistribute energy across neighboring pixels if spatial weights allow.

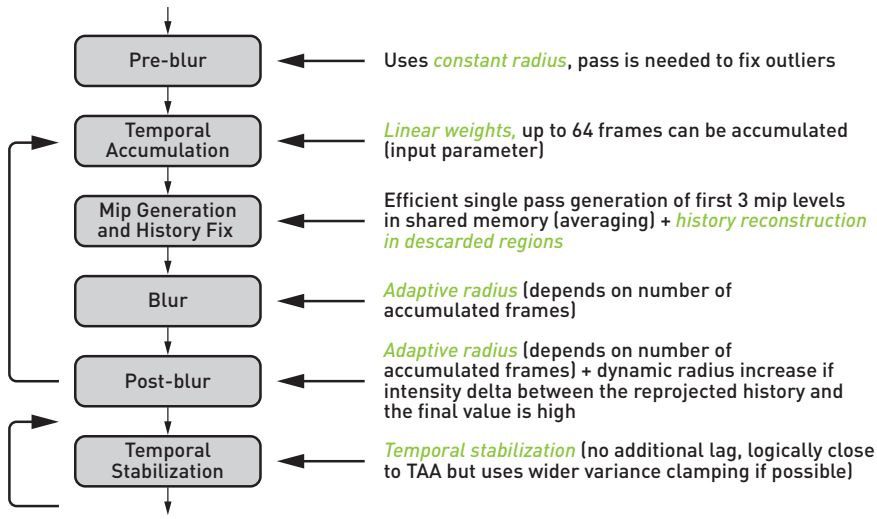


Figure 49-3. *Pipeline overview.*

Additionally, this pass handles checkerboarded half-resolution input. If the pixel center gets into a black cell (without valid data), the data gets reconstructed by applying bilateral filter to two candidates: left and right pixels. If this happens for a spatial sample, the algorithm shifts the sample to the left or to the right in a consecutive order.

49.5.2 TEMPORAL ACCUMULATION

The *temporal accumulation* pass increments the number of accumulated frames (`A`) per pixel, or sets it to 0 if a disocclusion is detected. Less noisy input after the pre-blur pass gets accumulated with the reprojected history buffer based on the number of accumulated frames using the formula:

$$\texttt{Output} = \texttt{lerp}\left(\texttt{History}, \texttt{Input}, \frac{1}{1+\texttt{A}}\right), \tag{49.1}$$

where `Input` represents the noisy input and `History` is the reprojected history.

It's used "as is" for diffuse signals that perfectly follow surface motion. Specular tracking is complicated; it implies computing two history candidates based on two types of reprojection (Section 49.8), but the same formula is used to mix them with the current input.

49.5.3 MIP GENERATION

For the accumulated diffuse, specular, and `viewZ` textures, ReBLUR generates the first four mip levels, which are used in the following history fix pass (Section 49.5.4) for hierarchical history reconstruction in regions with discarded history.

49.5.4 HISTORY FIX

In discarded regions in the history buffer, we have only a raw 1 rpp signal. Hence, we need to do something to avoid leaving noise in the final image. Using a very wide spatial filter is a good option, but it is expensive. Mip levels can be used to emulate a high rpp input in exchange for resolution:

Mip Level	Rays per Pixel	Data Tile
0 (base)	1 (real)	1 pixel
1	4 (virtual)	2×2 pixels
2	16 (virtual)	4×4 pixels
3	64 (virtual)	8×8 pixels

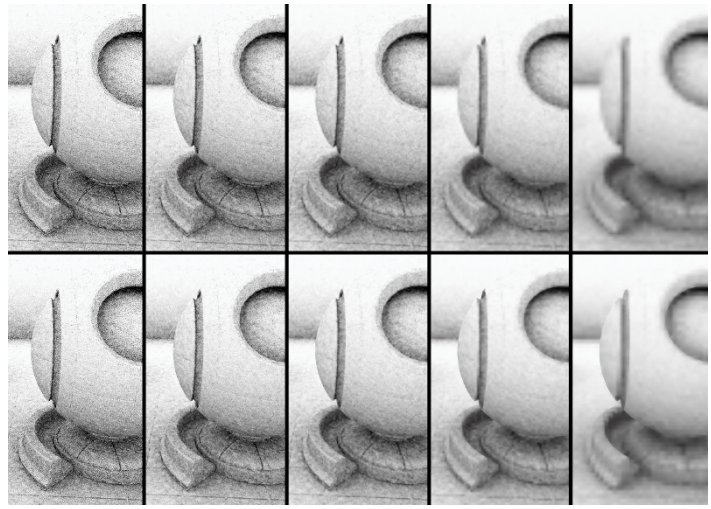


Figure 49-4. *History fix operation on an example of ambient occlusion. Top: first four mip levels of the input signal (the leftmost image is the input), upsampled using simple bilinear upsampling (a color bleeding effect is clearly visible). Bottom: the same four levels bilaterally upsampled (color bleeding free upsampling).*

History reconstruction starts with a coarse level (it depends on the roughness of the specular signals) and rapidly converges back to the base level. To avoid color bleeding, a hierarchical reconstruction filter is used. It applies a 3×3 cross-bilateral filter, starting from the coarsest mip level for the pixel roughness (Figure 49-4). For each sample in the footprint, its weight is computed by comparing real `viewZ` (from the base level) with the interpolated `viewZmip` (from the current mip level). If the sum of the weights is close to zero, the algorithm switches to the next finer level because the data on the current level is heavily averaged and doesn't match `viewZ` for the current pixel. This process repeats until the sum of weights is less than a threshold value.

The *history fix* pass doesn't use other weights (like normal and roughness) for performance reasons. Full reconstruction only happens for 1–3 frames (depending on roughness) and only in newly appeared regions; following after the history fix pass, the blur passes (Section 49.5.5) redistribute energy based on material properties. Even on the next frame after a history reset, previously blurred signals will be mixed with the current input in the temporal accumulation pass (Section 49.5.2), and the history fix pass will switch to a finer level.

Additionally, this pass performs variance-based color clamping, to faster tracked the history (with fewer accumulated frames) to increase overall response time, and anti-firefly filtering, which is based on the Rank Conditioned Rank Selection filter [1].

49.5.5 BLUR

The *blur* pass is the only spatial filter that uses a per-pixel stochastically rotated Poisson disk. The other filter passes use a statically defined Poisson disk, which is lined into the shader assembly by the compiler. This pass uses the same Poisson distribution, but applies a random rotation in each pixel. The rotation can be at a random angle, but ReBLUR uses animated in-time 4×4 Bayer dithering. To avoid introducing potentially dependent texture loads, per-pixel rotation is implemented using arithmetic logic unit (ALU) instructions only.

All blur passes look very similar. Each spatial filtering pass computes the blur radius depending on the number of accumulated frames, material roughness, and hit distance (the maximum blur radius is a user-controllable parameter). Additionally, blur passes set up the geometry basis (Section 49.9), because sampling is done in world space, and the parameters for geometry, normal, roughness, and hit distance weights to be used during sampling. At the end, cross-bilateral filtering is applied using a Poisson disk with eight samples (this low sample count has been selected to provide a good balance between image quality and performance).

49.5.6 POST-BLUR

The previous pass is not enough for spatial filtering, so a *post-blur* pass has been designed to perform cleanup after the blur pass. It does exactly the same thing but with one big difference: the blur radius is scaled proportional to the difference between the output of the blur pass (Section 49.5.5) and of the temporal accumulation pass (Section 49.5.2). The result of the post-blur pass will be used as the history buffer in the next frame. This dynamic adaptation of the filter radius allows us to minimize color divergence due to recurrent blurring.

49.5.7 TEMPORAL STABILIZATION

ReBLUR uses only one temporal accumulation pass. The *temporal stabilization* pass is different. It's main purpose is to stabilize the denoiser

output. It has its own history, which never goes to the temporal feedback loop. Logically, it's very close to temporal supersampling with variance color clamping [5]. Temporal stabilization of the specular signal requires accurate reprojection, which can't be achieved using surface motion only. Like in the temporal accumulation pass, two types of reprojection are used to get history candidates (Section 49.8), and the interpolation factor between the two is known and comes from the temporal accumulation pass (Section 49.5.2).

Another very important task for this pass is anti-lag handling. ReBLUR has been designed to do accurate reprojection of diffuse and specular signals from the previous frame to the current frame even with slow accumulation factors, but there should be something reacting on rapid changes in the environment not related to the camera movement, like handling dynamic objects or dynamic lights.

49.6 DISOCCLUSION HANDLING

Regardless of the signal type, the number of accumulated frames will be set to 0 in the temporal accumulation pass if a disocclusion is detected. A disocclusion is a new portion of a scene in a pixel, which was occluded in the previous frame, i.e., it doesn't exist in the history buffer. It makes the latter invalid for this pixel. Disocclusion can be detected by measuring the plane distance between the plane $\{X_{prev}, \mathbf{n}\}$ and another previous position, reconstructed from the previous frame. The resulting plane distance is normalized by the distance between the camera and the point and is compared with some small threshold value (around 0.5–1.5%). For a 2×2 bilinear footprint, disocclusion values can be computed as shown in Listing 49-2.

It's worth noting that $\mathbf{n}$ and $\mathbf{n}_{prev}$ can be averaged inside corresponding 2×2 bilinear footprints. It was discovered that this worked better for silhouettes, especially if a pixel on it had a normal coplanar to the screen normal. Using geometric normals instead of per-pixel normals can slightly improve image quality, but it becomes less relevant if averaged normals are used. The presented method is suited well for subpixel jittered inputs.

49.7 DIFFUSE ACCUMULATION

Diffuse accumulation is done by reprojecting accumulated history back to the current frame using a bicubic Catmull–Rom filter to avoid blurring under slow pixel motion. This filter has a higher degree and requires disocclusion

Listing 49-2. *Disocclusion handling.*

```
// N = surface normal for the current frame
// Nprev = surface normal for the previous frame
// prevViewZ = viewZ in the 2x2 footprint for the previous frame
float4 ComputeDisocclusion2x2( float threshold, float isInScreen, float4x4
      mWorldToViewPrev, float3 Xprev, float3 N, float3 Nprev, float
      invDistToPoint, float4 prevViewZ )
{
    // Out-of-screen = occlusion
    threshold = lerp(-1.0, threshold, isInScreen);

    float NoXprev1 = abs( dot( Xprev, N ) );
    float NoXprev2 = abs( dot( Xprev, Nprev ) );
    float NoXprev = max( NoXprev1, NoXprev2 ) * invDistToPoint;
    float Zprev = mul( mWorldToViewPrev, float4( Xprev, 1.0 ) ).z;
    float NoVprev = NoXprev / abs( Zprev );
    float4 relativePlaneDist = abs(NoVprev * abs(prevViewZ) - NoXprev);

    return step( relativePlaneDist, threshold );
}
```

detection in a 4×4 region. Bicubic filtering is used only if the entire footprint is valid; otherwise, we revert to a 2×2 bilinear footprint. Bilinear filtering uses custom weights to avoid using rejected samples, e.g., samples found during disocclusion handling or samples with backfacing normals (the plane distance is close to 0, but the normal is oriented in the opposite direction). If at least one sample in the footprint is valid for the current frame, the number of accumulated frames gets increased by 1, i.e., `A += 1`. Additionally, a final checkerboard resolve takes place at this stage.

49.8 SPECULAR ACCUMULATION

Specular accumulation does the same steps as in diffuse accumulation: disocclusion tracking, advancing the number of accumulated frames, and checkerboard resolve. But because specular reprojection can't be tracked using surface motion in a general case, specular reprojection from the previous frame is a combination of surface and virtual motion–based reprojections in ReBLUR.

49.8.1 SURFACE MOTION–BASED SPECULAR REPROJECTION

In this case regular surface motion vectors are used to reproject specular history from the previous to the current frame. If the accumulation factor is a predefined constant, it won't work for low/moderate roughness, glancing viewing angles, and highparallax. *Parallax* is the ratio between the camera

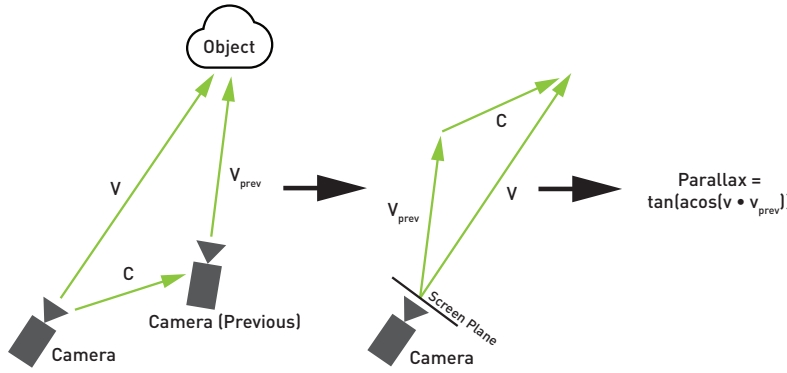


Figure 49-5. *Parallax between the current and previous positions of a point between two frames.*

movement vector projection and the distance to the point. In other words, parallax shows how far the new viewing direction is from the previous viewing direction (Figure 49-5).

One important property of parallax is that it's 0 for the entire frame if the camera is not moving and only rotating. In this case we can reproject the previous frame (if the world is static) without any artifacts, because simple reprojection can handle this mapping, preserving the viewing vector.

Parallax can be computed using the method in Listing 49-3. It can be used to estimate the confidence of surface-based specular motion (regular MVs) in two ways:

- Adjust the accumulation speed (larger parallax implies faster accumulation).
- Clamp the reprojected history based on parallax (larger parallax implies more clamping).

Listing 49-3. *Compute parallax.*

```
float ComputeParallax( float3 X, float3 Xprev, float3 cameraDelta )
{
    float3 V = normalize(X);
    float3 Vprev = normalize(Xprev - cameraDelta);
    float cosa = saturate( dot(V, Vprev) );
    float parallax = sqrt( 1.0 - cosa * cosa ) / max(cosa, 1e-6);
    parallax *= 60; // Optionally normalized to 60 FPS

    return parallax;
}
```

Listing 49-4. *Estimate specular accumulation.*

```
// MAX_ACCUM_FRAME_NUM = 32-64
// SPEC_ACCUM_BASE_POWER = 0.5-1.0 (greater values lead to less aggressive accumulation)
// SPEC_ACCUM_CURVE = 1.0 (aggressiveness of history rejection depending on viewing angle: 1 = low, 0.66 = medium, 0.5 = high)
float GetSpecAccumSpeed(float Amax, float roughness, float NoV, float parallax)
{
    float acos01sq = 1.0 - NoV; // Approximation of acos^2 in normalized
                                // form
    float a = pow(saturate(acos01sq), IsReference() ? 0.5 : SPEC_ACCUM_CURVE);
    float b = 1.1 + roughness * roughness;
    float parallaxSensitivity = (b + a) / (b - a);
    float powerScale = 1.0 + parallax * parallaxSensitivity;
    float f = 1.0 - exp2(-200.0 * roughness * roughness);
    f *= pow(saturate(roughness), SPEC_ACCUM_BASE_POWER * powerScale);
    float A = MAX_ACCUM_FRAME_NUM * f;

    return min(A, Amax);
}
```

The latter is not used in ReBLUR because the pre-blur spatial filtering pass is not enough to guarantee sufficient noise reduction of the output signal. High local variance prevents us from using wide variance color clamping in the temporal accumulation pass (but it's still a good option if the input signal is relatively clean).

Now we can define a function A = *f*(A, **n** · **v**, parallax) that estimates the maximum number of allowed accumulated frames without increasing temporal lag. The function *f* should have the following properties:

- *f* is smaller for low roughness.
- *f* is smaller for small absolute values of **n** · **v** (the Fresnel effect does not follow surface motion).
- *f* is smaller for high parallax, i.e., when the viewing angle changes significantly between two frames.

We found empirically that the method in Listing 49-4 works well in practice.

Finally, we can combine all the parts into a method for reprojecting and combining history with the current signal using surface motion only:

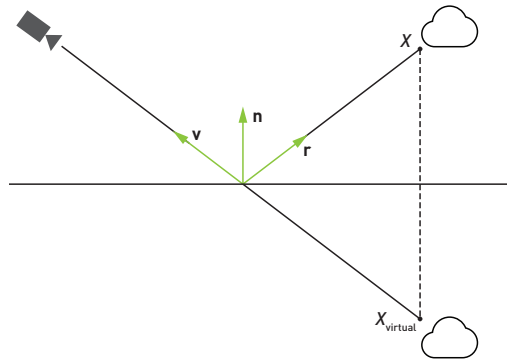


Figure 49-6. *Reflection virtual position.*

1. `parallax = ComputeParallax( X, Xprev, cameraDelta )` (Listing 49-3).
2. Estimate allowed maximum number of accumulated frames: `Asurf = GetSpecAccumSpeed( A, roughness, NoV, parallax )` (Listing 49-4).
3. Reproject specular history using surface motion.
4. `current = lerp( history, input, 1 / ( 1 + Asurf ))`.

49.8.2 VIRTUAL MOTION–BASED SPECULAR REPROJECTION

Virtual motion–based reprojection is a trick that works for mirror reflections. It's based on the fact that the reflected world has its own motion, based on the motion of the reflection virtual position (Figure 49-6).

In the shader code, projection to the previous frame based on virtual motion can be computed as (assuming that **v** points out of the surface):

```
1 float3 Xvirtual = X - V * hitDist;
2 float2 pixelUvVirtualPrev = GetScreenUv( gWorldToClipPrev, Xvirtual );
```

The resulting texture coordinates are used to retrieve the reprojected history.

If used "as is," this method doesn't work for anything except pure mirrors (roughness = 0), because in a general case the specular lobe can be "flattened," making reflections "visually" much closer to the surface. Logically, virtual motion should resolve to surface motion for roughness = 1 (Figure 49-7). Mathematically, a good approximation of this process can be

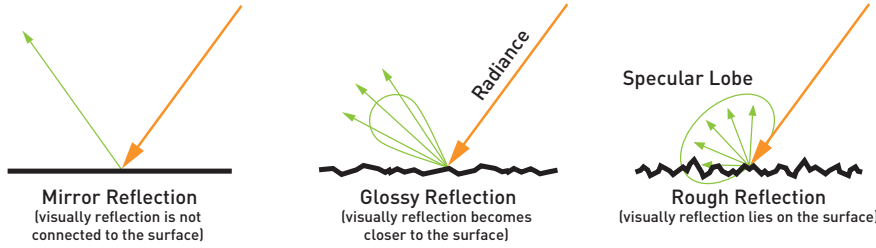


Figure 49-7. *Roughness and specular reflections.*

Listing 49-5. *Specular dominant factor.*

```
float GetSpecularDominantFactor(float NoV, float roughness)
{
    float a = 0.298475 * log(39.4115 - 39.0029 * roughness);
    float f = pow(saturate(1.0 - NoV), 10.8649)*(1.0 - a) + a;

    return saturate(f);
}
```

calculations based on the specular dominant direction, which define how close the specular dominant direction $\mathbf{d}_{GGX}$ is to the normal vector **n** or the reflection vector **r** (Listing 49-5). See Lagarde and de Rousiers [2, page 69] for more details about the implementation of the function `GetSpecularDominantFactor`.

Now we can adjust the virtual position calculation as shown in Listing 49-6. As you can see, in contrast to the previous "vanilla" method, we have introduced a scaling factor (`f`), which says how close the virtual position should be to the surface position based on the current roughness, normal, and view vector. The virtual position gets resolved to the surface position if the factor is close to 0 (i.e., $\mathbf{d}_{GGX}$ is close to **n** and roughness is high). But even after this addition, virtual motion still leaves a lot of artifacts and ghosting

Listing 49-6. *Virtual position in the reflected world.*

```
float3 GetXvirtual(float3 X, float3 V, float NoV, float roughness, float
    hitDist)
{
    float f = GetSpecularDominantFactor(NoV, roughness);
    return X - V * hitDist * f;
}
```

Listing 49-7. *Virtual motion confidence.*

```
// Out of screen
float isInScreenVirtual = float(all(saturate(pixelUvVirtualPrev) ==
    pixelUvVirtualPrev));
float virtualHistoryConfidence = isInScreenVirtual;

// Normal
virtualHistoryConfidence *= GetNormalWeight(roughness, N,
    prevNormalVirtual);

// Roughness
virtualHistoryConfidence *= GetRoughnessWeight(roughness,
    prevRoughnessVirtual);

// Hit distance
float hitDistMax = max(hitDistVirtual, hitDist);
float hitDistDelta = abs(hitDistVirtual - hitDist) / (hitDistMax + viewZ
    );
float thresholdMin = 0.02 * smoothstep(0.2, 0.01, parallax);
float thresholdMax = lerp(0.01, 0.25, roughness * roughness) +
    thresholdMin;
virtualHistoryConfidence *= smoothstep(thresholdMax, thresholdMin,
    hitDistDelta);
```

because we should account for various types of disocclusion happening in the reflected world or other factors leading to such a disocclusion. There are several, and all of them are multiplied into a single aggregative parameter called *virtual motion confidence* (Listing 49-7).

This confidence is used in two ways: it affects how heavily the reprojected history will be clamped to the input signal using variance color clamping, and additionally it accelerates history accumulation if confidence is low.

Hit distance–based confidence correction requires relatively clean hit distances; this is why ReBLUR denoises them as well. Input hit distances can be too noisy to be used "as is." Instead, the algorithm mixes input hit distance with surface-based parallax-corrected reprojection (fast accumulated) to reduce the noise level.

The specular dominant factor is used to mix surface motion–based and virtual motion–based reprojections. That's the second ingredient, which can be called the *amount of virtual motion*.

Becuase virtual motion is correct only for flat surfaces, local curvature can be used to solve the lens equation and adjust the virtual point position and/or amount of virtual motion.

49.8.3 COMBINED SOLUTION

Now we can assemble all parts into a combined solution of reprojecting the specular history texture from the previous frame (Listing 49-8). An example of the method applied in *Cyberpunk 2077* can be seen in Figure 49-8.

Listing 49-8. *Combined solution for specular accumulation.*

```
// Reproject history using surface motion.
float2 uvPrev = GetScreenUv(gWorldToClipPrev, Xprev);
float4 historySurf = ApplyBilinearFilterWithCustomWeights(historyTexture,
     uvPrev, occlusion2x2); // (Listing 49-2)

// Find surface motion-based accumulation speed (Listing 49-4).
float Asurf = GetSpecAccumSpeed(A, roughness, NoV, parallax);

// Combine surface motion-based reprojected
// history with the noisy input.
float Ahitdist = min(Asurf, maxHitDistAccumulatedFrameNum);
float4 currentSurf;
currentSurf.xyz = lerp(historySurf.xyz, input.xyz, 1/(1 + Asurf));
currentSurf.w = lerp(historySurf.w, input.w, 1/(1 + Ahitdist));

// Find Xvirtual (Listing 49-6).
float3 Xvirt = GetXvirtual(X, V, NoV, roughness, currentSurf.w);

// Reproject history using virtual motion.
float2 uvPrevVirt = GetScreenUv(gWorldToClipPrev, Xvirt);
float4 historyVirt = ApplyBilinearFilter(historyTexture, uvPrevVirt);

// Estimate virtual motion confidence (Listing 49-7).
float confidence = isInScreenVirtual;
confidence *= GetNormalWeight(...);
confidence *= GetRoughnessWeight(...);
confidence *= GetHitDistanceWeight(...);

// Compute amount of virtual motion.
float amount = GetSpecularDominantFactor(NoV, roughness);
amount *= isInScreenVirtual;
amount *= GetNormalWeight(...); // optional

// Apply optional wide variance clamping to the virtually
// reprojected history if confidence is low.
float4 historyVirtClamped = ApplyVarianceClamping(historyVirt);
float f = lerp(confidence, 1, roughness * roughness);
historyVirt = lerp(historyVirtClamped, historyVirt, f);

// Find virtual motion-based accumulation speed.
float Avirt = GetSpecAccumSpeed(A, roughness, NoV, 0);

// Adjust virtual motion-based accumulation speed for low confidence.
float Amin = min(Avirt, MIP_NUM * sqrt( roughness )); // MIP_NUM = 3-4
float a = lerp( 1/(1 + Amin), 1/(1 + Avirt), confidence );
Avirt = 1.0 / a - 1.0;

```

```
47 // Combine virtual motion-based reprojected history
48 // with the noisy input.
49 float Ahitdist = min(Avirt, maxHitDistAccumulatedFrameNum);
50 float4 currentVirt;
51 currentVirt.xyz = lerp(historyVirt.xyz, input.xyz, 1/(1 + Avirt));
52 currentVirt.w = lerp(historyVirt.w, input.w, 1/(1 + Ahitdist));
53
54 // Mix surface and virtual motion-based combined specular
55 // signals into the final result.
56 float4 currentResult = lerp(currentSurf, currentVirt, amount);
57
58 // Mix surface and virtual motoin-based numbers of accumulated
59 // frames into a single value for the next frame.
60 float a = lerp( 1/(1 + Asurf), 1/(1 + Avirt), amount );
61 float Acurr = 1.0 / a - 1.0;
```



Figure 49-8. *Cyberpunk 2077 with ray tracing. Left: before denoising. Right: after denoising.*

49.9 SAMPLING SPACE

ReBLUR gathers image samples based on a world-space kernel. For diffuse signals the sampling basis is constructed around the pixel normal. For specular signals the basis is constructed around the reflected $\mathbf{d}_{GGX}$ (Figure 49-9). The tangent vector is perpendicular to the pixel normal, which allows us to scale it based on the viewing angle to achieve realistic specular elongation. The code snippet in Listing 49-9 computes the plane at which the Poisson disk samples are distributed.

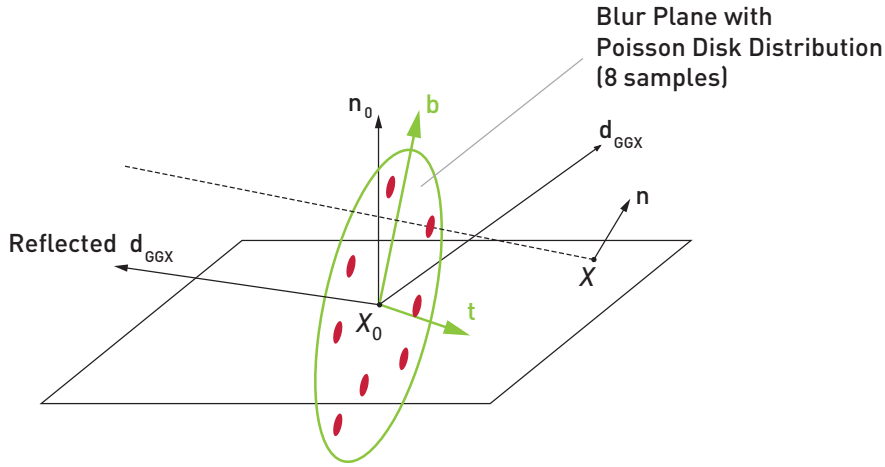


Figure 49-9. *Sampling space.*

Listing 49-9. *Get kernel basis.*

```
float2x3 GetKernelBasis( float3 V, float3 N, float roughness = 1.0 )
{
    float3x3 basis = ConstructBasis( N );
    float3 T = basis[ 0 ];
    float3 B = basis[ 1 ];

    float NoV = abs( dot( N, V ) );
    float f = GetSpecularDominantFactor( NoV, roughness );
    float3 R = reflect( -V, N );
    float3 D = normalize( lerp( N, R, f ) );
    float NoD = abs( dot( N, D ) );

    if( NoD < 0.999 && roughness != 1.0 )
    {
        float3 Dreflected = reflect( -D, N );
        T = normalize( cross( N, Dreflected ) );
        B = cross( Dreflected, T );

        float NoV = abs( dot( N, V ) );
        float acos01sq = saturate( 1.0 - NoV );
        float skewFactor = lerp( 1.0, roughness, sqrt( acos01sq ) );
        T *= skewFactor;
    }

    return float2x3( T, B );
}
```

49.10 SPATIAL FILTERING

Our spatial filters use Poisson disk stochastic sampling with eight taps. The final sample weight w_{sample} in our spatial filtering is a product of several

weights:

$$w_{sample} = w_{geometry} \cdot w_{normal} \cdot w_{roughness} \cdot w_{hitDist}, \tag{49.2}$$

where

- $w_{geometry}$ is how close the current sample position is to the plane $\{X, \mathbf{n}\}$ at the kernel center [7].
- w_{normal} is the cutoff parameters based on the average specular lobe angle.
- $w_{roughness}$ allows us to choose lobes with similar spread angles (for specular only).
- $w_{hitDist}$ allows us to avoid over-blurring in regions with contrast hit distances (contact and far reflections).

Any weight w is computed using the following formula:

$$w = \text{smoothstep}(0, 1, \text{saturate}(1 - |ax + b|)), \tag{49.3}$$

where x is the value for which we compute a weight (e.g., for the normal vector, x is a dot product with the normal at the center and the normal at the current tap: $x = \mathbf{n}_{tap} \cdot \mathbf{n}_{center}$). The scalars a and b represent the parameterization parameters: a controls the normalization and b controls the horizontal shift of the curve.

Most of the parameterization parameters depend on the number of accumulated frames and become more strict over time. It's worth noting that normal weight normalization depends on the specular lobe angle, which is computed using the following formula:

```
float GetSpecularLobeHalfAngle( float linearRoughness, float percentOfVolume
    = 0.75 )
{
    float m = linearRoughness * linearRoughness;
    return atan( m * percentOfVolume / ( 1.0 - percentOfVolume ) );
}
```

See Lagarde and de Rousiers [2, page 72] for more details about the `GetSpecularLobeHalfAngle` implementation.

49.11 ANTI-LAG

The temporal stabilization pass has its own history. It's slower than the history in the temporal accumulation pass before it gets clamped to the

current frame. Anti-lag is based on a relative delta between slow (temporal accumulation) and even slower (temporal stabilization) histories. There are two types of anti-lag calculations: hit distance–based and intensity-based. If the delta is high, the number of accumulated frames will be reduced, and on the next frame accumulation will continue from this new value.

49.12 LIMITATIONS

ReBLUR works with pure radiance coming from a particular direction. Spatial filtering can redistribute this energy across neighboring pixels. This means that if irradiance is passed instead (i.e., material information is included), then, as the result, ReBLUR can blur out many details. To overcome this problem, BRDF materials should be applied after denoising.

ReBLUR spatial filtering is based on the specular dominant direction. It makes, for example, diffuse denoising be a special case of specular denoising for roughness = 1. It implies that ReBLUR is less accurate for materials such as non-Lambertian diffuse materials, specular materials with retroreflection, anisotropic specular materials (but it's not difficult to extend the algorithm to support it), multi-lobe specular materials, or multi-layered materials. It's worth noting that multi-layered (or multi-lobe) materials can be denoised per layer (per lobe) in the current implementation.

49.13 PERFORMANCE

There are three variants of the denoiser: diffuse only, specular only, and combined diffuse-specular. The following table shows per-pass performance of the combined version at 2560 $\times$ 1440 native resolution measured on RTX 3090:

Pass	**Runtime (ms)**
Pre-blur	0.48
Temporal accumulation	0.59
Mip generation	0.10
History fix	0.30
Blur	0.40
Post-blur	0.44
Temporal stabilization	0.41
Total:	**2.72**

49.14 FUTURE WORK

For future work, we would like to add support for anisotropic roughness and to look into alternative ways of deriving virtual motion vectors, which are needed for specular reflections. We also want to compare the Poisson disk sampling used in ReBLUR against Á-Trous, e.g., the hierarchical spatial filter used in ReLAX and SVGF [6], and to look into ways of combining the two approaches.

REFERENCES

[1] Hardie, R. C. and Barner, K. E. Rank conditioned rank selection filters for signal restoration. *IEEE Transactions on Image Processing*, 3(2):192–206, 1994. DOI: 10.1109/83.277900.

[2] Lagarde, S. and de Rousiers, C. Moving Frostbite to physically based rendering. SIGGRAPH Course, https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf, 2014.

[3] NVIDIA. NVIDIA real-time denoisers. https://developer.nvidia.com/nvidia-rt-denoiser, 2021.

[4] Panteleev, A. Rendering perfect reflections and refractions in path-traced games. *NVIDIA Developer Blog*, https://developer.nvidia.com/blog/rendering-perfect-reflections-and-refractions-in-path-traced-games/, August 13, 2020. Accessed February 10, 2021.

[5] Salvi, M. An excursion in temporal supersampling. Presentation at Game Developers Conference, https://www.dropbox.com/sh/dmye840y307lbpx/AAAQpC0MxMbuOsjm6XmTPgFJa?dl=0, 2016. Accessed February 10, 2021.

[6] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, 2:1–2:12, 2017. DOI: 10.1145/3105762.3105770.

[7] Shyshkovtsov, O., Archard, B., Karmalsky, S., and Zhdan, D. Exploring the ray traced future in Metro Exodus. Presentation at Game Developers Conference, https://www.gdcvault.com/play/1026159/Exploring-the-Ray-Traced-Future, 2019. Accessed February 10, 2021.

[8] Zhdan, D. Fast denoising with self stabilizing recurrent blurs. Presentation at Game Developers Conference, https://www.gdcvault.com/play/1026701/Fast-Denoising-With-Self-Stabilizing, 2020. Accessed February 10, 2021.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

CHAPTER 50

PRACTICAL SOLUTIONS FOR RAY TRACING CONTENT COMPATIBILITY IN UNREAL ENGINE 4

Evan Hart
NVIDIA

ABSTRACT

Though modern graphics hardware offers features and performance for real-time ray tracing, consumer applications like games must target legacy hardware to serve a large install base. This chapter addresses some challenges and solutions for working with ray tracing in this legacy content. In particular, it demonstrates a solution for mixing raster and ray traced translucency, as well as a set of solutions for animated foliage.



Figure 50-1. *A forest scene rendered with ray tracing in Unreal Engine 4 (UE4) using animated foliage. The scene uses ray tracing for shadows and reflections. The scene was authored by Richard Cowgill using the Forest - Environment Set by Nature Manufacture available through the UE4 Marketplace [4].*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_50

50.1 INTRODUCTION

Ray tracing opens up an impressive array of graphical effects that were previously challenging or impractical for real-time applications such as games. However, all consumer-focused applications like games have a heavy burden of legacy support. The effects and art must be authored to work well on the platforms that represent the vast majority of the install base when they release. This constraint demands that the primary focus of the development effort be on content tuned to work well for rasterization. These current market needs to prioritize rasterization during development limit the cycles available to tune specifically for ray tracing.

Due to the practical limitations of revising many man-years of asset development, the current path to enabling ray tracing effects in real-time games is to adapt the ray tracing effects to the raster-centric content already required. This chapter presents techniques for resolving challenges in two common rendering regimes: translucency and foliage. These techniques have been successfully used to enable ray tracing effects in shipping games.

Though the algorithms here were implemented in the context of the Unreal Engine, they are broadly applicable. NVIDIA's custom NvRTX branch of Unreal Engine 4 (available to anyone registered for Unreal Engine access through GitHub at https://github.com/NvRTX/UnrealEngine) demonstrates the implementation of these techniques and several more.

50.2 HYBRID TRANSLUCENCY

Translucency in games typically means anything that is not opaque. For rasterization, this implies alpha blending over other geometry. This category covers common objects like glass and water, but it also includes particle effects such as smoke and fire. Importantly, translucent objects bring special shading and composition challenges to raster graphics due to their many overlapping surfaces.

50.2.1 MOTIVATION

The standard implementation of ray traced translucency in Unreal Engine 4 (UE4) handles all translucent effects in a single pass. Primary rays are traced from the viewer to just in front of the closest opaque surface. The depth buffer provides this max ray distance. Rays shade the closest hit, including a reflection bounce and any necessary shadow rays, then fire a continuation ray.



(a) *Ray traced translucency*

(b) *Rasterized translucency*

Figure 50-2. *Ray traced translucency offers an impressive upgrade in shading. However, artists typically author the content with rasterization in mind. This scene is a variant of the Amazon Lumberyard Bistro [1].*

Depending on the settings and material type, the continuation ray potentially refracts. With the right settings and content, this algorithm produces a high-quality result. Translucent interactions are perfectly sorted, shading is substantially improved over shading used for raster translucency (Figure 50-2), and refraction effects are supported natively. Importantly, achieving this result requires that the content be configured appropriately. Without proper tuning, performance challenges and graphical artifacts commonly occur.

The most direct performance challenge with fully ray traced translucency comes from particle effects authored for rasterization. Particle systems often create volumetric effects with several sprites overlapping a single pixel. Capturing each layer with full ray traced translucency requires tracing an additional ray and one additional shading invocation. Though this may produce high-quality volumetric effects, the performance scales quite differently than originally intended by the artist. For example, even eight layers of ray tracing interactions are too few to render a simple fire and smoke particle system without clear hole artifacts where the rays terminate too early. More importantly, ray tracing a particle system at that level of overlap costs roughly 10 times the cost of rasterizing it. Although particle systems are the most common example to encounter this performance concern, the performance challenges are not limited to them alone. Artists may place dozens of small glasses inside a restaurant that they never intended to have a dramatic impact on the scene visuals. Due to the order-dependent nature of compositing, one cannot mix and match the

methods of rendering translucency. As fully ray traced translucency is an all-or-nothing affair, the only choices are to pay that cost for everything or to forgo the enhancements entirely.

Quality challenges with fully ray traced translucency are more varied than those of performance issues. Particles again show up at the top of the list of concerns. In UE4, first among the particle challenges is that much legacy content relies on a particle system referred to as *Cascade*, as opposed to a newer system named *Niagara*. The older Cascade system completely lacks ray tracing support, so enabling full ray traced translucency causes all particles from Cascade to completely disappear. This forces a developer using the legacy particle system to choose between re-authoring those systems or foregoing ray traced translucency. Even when all particle systems are supported, the authoring process of treating them as screen-aligned billboards may or may not hold up well under the different rendering methodology.

The next challenge for fully ray traced translucency comes from refraction and distortion. Raster translucency generally only offers distortion as a special effect carefully placed in the scene. Ray traced translucency accomplishes this distortion naturally through refraction. This requires that either refraction is disabled or all materials are configured properly with their index of refraction. The result is a choice between no distortion/refraction or editing potentially hundreds of materials as well as geometry to ensure that refraction properties have all been properly configured. Misconfigured refraction parameters are quite common in content that has been developed without the intention of supporting ray tracing from the start. A simple example is that of a window. Games will generally use a two-sided quadrilateral for the window. Even if the index of refraction is correct to drive the physically based rendering used by the engine, the lack of a second quadrilateral representing the backside of the glass will result in distortion. The ray will continue in the refracted direction as if looking into a solid block of glass. Finally, the best performance-tuning parameter for fully ray traced translucency presents its own image artifact challenges. Ray traced translucency requires that users place a limit on the number of layers traced. Though this greatly helps performance in cases of high depth complexity, the translucent surfaces beyond the limit are simply not rendered. Although the front one or two layers will have the most important impact on the image, having objects completely disappear is unfortunate (Figure 50-3).



(a) *Multi-layer ray traced translucency* **(b)** *Single-layer ray traced translucency*

Figure 50-3. *With only a single layer of translucency, overlapping glasses result in the glasses farther from the viewer disappearing in the overlapping sections.*

50.2.2 OUR HYBRID APPROACH

The hybrid translucency solution described here bridges the gap between ray traced translucency and raster translucency. Effects best suited for rasterization can use rasterization, while ray traced shading is applied to more important surfaces like window panes. Hybrid translucency fixes the ordering problem described previously by having the ray tracing pass place its results in an offscreen cache. The cache captures the radiance for translucent layers independently, rather than compositing them. This allows the rasterization phase of translucency to look up the shading result in the cache and substitute it for the shading value produced via rasterization. Importantly, all translucent primitives that wish to receive ray tracing effects are rendered twice (once via ray tracing and once via rasterization). As rasterization is used to composite everything, hybrid translucency is unable to provide the order-independent translucency benefit of fully ray traced translucency. However, hybrid translucency is intended to work with legacy content, which has already had to deal with this challenge. Further, hybrid translucency uses the same refraction and distortion methods as rasterization. Although those effects will not look any better than they did under rasterization, they will now function identically without the need for an artist to tune the effect specially for ray tracing. The result is a technique that functions and performs in a manner consistent with the rasterization systems for which the content has already been tuned, while offering greatly improved shading on surfaces like windows at little cost to the asset pipeline (Figure 50-4).

Hybrid translucency's ray traced shading cache functions by storing samples in a screen-space texture array. Each pixel in the framebuffer contains up to



(a) *Fully ray traced translucency* **(b)** *Hybrid translucency with two layers*

Figure 50-4. *Hybrid translucency replicates the most important visual enhancements of full ray tracing while continuing to support rasterized elements.*

N layers of shaded translucent surfaces (or events). Each event stores the reflected irradiance as well as the distance from the viewpoint. Importantly, it does not composite any transmission through the surface. The transmission is applied by the blending operations during compositing. Creating the cache is accomplished by a simple modification to the standard algorithm for handling ray traced transparency. Each pixel traces a ray from the eye into the scene with the maximum distance limited by the opaque geometry already written to the framebuffer. The closest hit is shaded and written to the first layer of the cache, then the ray is stepped forward and traced again to record additional events if necessary. The pseudocode in Listing 50-1 demonstrates the basic algorithm for the ray generation shader.

Once the shading cache is created, the ray traced results must be composited into the scene with the raster transparency. Compositing is performed in the standard rasterization order. The shader for translucent materials is enhanced with code to check the shading cache. The check first examines layer 0 to see if any ray traced data was written for the pixel. If so, it compares the distance from the viewpoint against what was stored in the cache. If the distance matches within a threshold, the shading data from the cache is used. To prevent paying the cost of searching with every translucent pixel, objects not supporting ray tracing skip the search by using a uniform branch. Listing 50-2 contains code for applying the data from the cache.

Importantly, the compositing pass for hybrid translucency maintains the shading functionality that would have otherwise applied. This allows for the production of reasonable results, no matter how many layers of translucency

Listing 50-1. *Ray generation pseudocode.*

```
GBufferData OpaqueData = ReadGBufferForPixel();

FarPosition = ReconstructionPosition(OpaqueData);

RayDescription Ray;
Ray.Origin = ViewerPosition;
Ray.Direction = normalize(FarPosition - ViewerPosition);
Ray.MinT = 0.0;
Ray.MaxT = length(FarPosition - ViewerPosition) - Epsilon;

for(int Event = 0; Event < MaxEvents; Event++)
{
    Payload HitData = Trace(Ray);
    if(!HitData.IsHit)
    {
        Break;
    }
    else
    {
        Color = ShadeHit(HitData);
        Distance = HitData.Distance;
        RecordLayer(Event, Color, Distance);
    }

    // Step forward for next search.
    Ray.MinT += HitData.Distance + Epsilon;
}
```

Listing 50-2. *Translucent cache search pseudocode.*

```
Color = ComputeShadingForTranslucentPixel();

if (SupportRayTracing)
{
    // Check if any layers were captured.
    if (Layers[pixel][0].Distance > 0)
    {
        Distance = Length(WorldPosition - ViewPosition);

        for(Layer = 0 : NumLayers - 1)
        {
            LayerDist = Layers[Pixel][Layer].Distance;
            Delta = abs(Distance - LayerDistance);

            if (Delta/Distance < Threshold)
            {
                // Surface matches the cached data.
                Color = Layers[pixel][Layer].Color;
                break; // Exit the loop.
            }
        }
    }
}
```



(a) *Two-layer hybrid translucency* **(b)** *Single-layer hybrid translucency*

Figure 50-5. *Hybrid translucency gracefully falls back to the raster effect, on which the content already needs to rely for most systems.*

are present. The top *N* layers will receive ray traced shading, while the deeper layers will simply fall back to the effect as it would have been with rasterization (Figure 50-5). As the topmost layer will provide the most significant contribution to the image, ray tracing only a single layer of translucency is typically enough.

50.2.3 RELATIONSHIP TO ORDER-INDEPENDENT TRANSPARENCY

The hybrid translucency approach described in this chapter explicitly avoids solving the problem of the order of translucent surfaces, but it does share some similarities. Depth peeling [3] converts translucency into layers to allow compositing in depth order, and A-buffers [2] produce sorted per-pixel lists of transparent surfaces. These concepts relate to the ordered sample cache used by this hybrid translucency algorithm. In contrast to these order-independent transparency (OIT) algorithms, the structure only serves to ensure that the closest events were captured and to optimize cache lookups. Importantly, this hybrid algorithm does not preclude the use of OIT approaches during the compositing pass. It relies on whichever one is in use for compositing standard raster transparency, with the key requirement that a single method be used to order both the raster and the ray traced components.

50.2.4 PERFORMANCE

As with all advanced visual effects, managing and scaling performance is important with hybrid translucency. As discussed previously, hybrid translucency can cut the amount of shading both by reducing the objects considered for translucent ray tracing and by restricting the number of layers



(a) *Ray traced translucency with two refraction events: 16.2 ms*

(b) *Half-resolution hybrid translucency with only the top layer traced: 4 ms + 1.5 ms*

Figure 50-6. *Hybrid translucency produces overall better image quality at one third of the cost in this scene. Notice that much of the glassware is not completely visible in the purely ray traced version due to exhausting its refraction event count. All timings were taken on an RTX 3090 at* 1920×1080

captured. The ability to degrade more gracefully is the soul of the hybrid translucency's performance advantage over standard ray traced translucency. The tracing pass for hybrid translucency is effectively identical in cost when comparing equal numbers of events shaded. Shading two layers of refraction rays in a complex test scene shows a cost of 16.1 milliseconds (ms) for pure ray tracing and 16.2 ms for hybrid ray tracing (Figure 50-6a). The hybrid method also requires an additional 1.5 ms to rasterize the translucency. (All tests performed at 1920×1080 resolution on an RTX 3090.) However, the ray traced scene suffers from translucent objects disappearing even with two layers of events. The hybrid scene has no objects disappearing, and it can reduce the cost to a single layer with hardly any visual impact, reducing its cost to 7.9 ms. Additionally, the shading cache layers of hybrid translucency can be rendered at half the resolution in a checkerboard or interleaved style while upsampling at compositing time. As expected, halving the number of samples halves the cost of ray tracing, reducing the cost to 4 ms for this test case (Figure 50-6b).

Finally, all ray traced translucency can benefit from using rasterization to compute a mask of potentially transparent pixels on the screen. Rasterizing the few translucent objects that participate in ray tracing against the depth buffer very quickly marks which pixels can ever produce a translucency hit. This allows the ray generation shader to terminate without firing a ray for regions with no coverage. For most scenes, this saves hundreds of thousands of rays cast for what is often under one tenth of a millisecond in cost. However, the benefit depends on the amount of translucency in the scene, so the utility of this technique will vary.

50.3 FOLIAGE

Foliage is an integral part of outdoor environments in nearly all games. The term *foliage* in the context of games covers everything from large trees to grass. In UE4, as in most games, the majority of foliage is handled through somewhat specialized systems to allow the high density typically desired. This system replicates dozens to hundreds of identical copies, commonly called *instances*, of the meshes with different transformations to produce a rich environment. On top of the varied static transforms, foliage systems typically utilize some form of vertex shader animation. Though it may be as simple as a sine wave to sway the grass back and forth, this ambient motion brings life to a scene.

50.3.1 REPRESENTING ANIMATED FOLIAGE

Data management is the key issue with supporting foliage in a ray tracing context for games. Exploiting the instanced nature of foliage is the first step in managing the costs. Placing foliage as instances into the top-level acceleration structure (TLAS) with shared entries in the shader binding table provides the solution to manage the costs associated with thousands of objects. A modest forest scene in UE4 spends over 6 ms of CPU time setting up ray tracing instances that are setup independently, as opposed to 0.6 ms for using shared setup. Importantly, it may even be worthwhile to forego multiple levels of detail, as the costs associated with managing the multiple levels of detail may outweigh the gains. One aspect to managing the instances is efficiently culling to ensure that only the relevant instances are processed. In general, ray tracing makes the culling problem more difficult, as reflection rays are harder to account for. A good solution is to cull by projecting the bounding sphere to a solid angle from the viewpoint. A culling angle of 1–2 degrees ensures that the screen area impacted by the object is respected across all but the most extreme regimes. Tall trees will accurately cast their long shadows and be prominent in reflections, while the impact of the thousands of small tufts of grass will be minimized.

The ambient motion of foliage substantially magnifies the data management issue while also introducing costs of its own. Bottom-level acceleration structures (BLASs) must be created uniquely for each different deformation. This means that the simple vertex shaders used to add ambient motion to all the foliage in a scene are producing thousands of unique meshes each from the perspective of ray tracing. The naive solution requires running a compute

shader over each instance to update the vertices, then refitting or rebuilding the BLAS each frame. A test of a simple forest scene in UE4 shows that even capping this processing at 256 instances per type costs over 50 ms of GPU processing on an NVIDIA RTX 3090. Such a solution is untenable in a real-time application. A more practical fallback is to simply skip the ambient motion and leave all instances in their neutral poses. Placing just the static objects instanced into the TLAS produces a very reasonable representation for many gaming scenarios. It ensures that the foliage is accurately represented in size and approximate location. Importantly, the ray tracing representation is only observed as part of secondary effects. Reflections will show correctly placed and lit foliage. Without close inspection, the lack of motion will frequently be missed, as large near-perfect mirror reflections are uncommon. Though occlusion effects like shadows and ambient occlusion may often get by with a lack of motion especially when casting on moving objects, the degree to which it is acceptable will vary based on the content. The appearance of stationary leaf shadows on static objects will sometimes stand out as objectionable, especially when the shadows created via shadow maps move. Importantly, fully accurate motion, which would require the 50 ms cost mentioned previously, is not typically necessary. As long as shadows have motion generally consistent with the behavior of the foliage as seen by the viewer, a convincing effect is possible.

Reusing animations across multiple instances allows the shadows to have behavior consistent with the motion prescribed for the foliage, while not paying the cost for matching the animations exactly. Sharing the animations between instances is a simple extension to the simulations used for stand-alone deformed meshes. The only additional effort required is that the result must keep the same neutral coordinate frame as the original mesh. The translations, rotations, and scales for the instances still apply to produce a convincing animated instance in the world. Minimizing the number of simulated instances is important, as each additional simulated instance is work and limits reuse. Often, convincing results are achievable with only a single simulated instance for each distinct foliage mesh in a forest. Again, the user never views the ray traced results directly.

50.3.2 INEXACT OCCLUSION

Though purely static foliage or replicated animations do a reasonably good job on reflections and shadow casting from foliage onto other objects, self-occlusion effects of foliage require a bit more care. Because the vertex



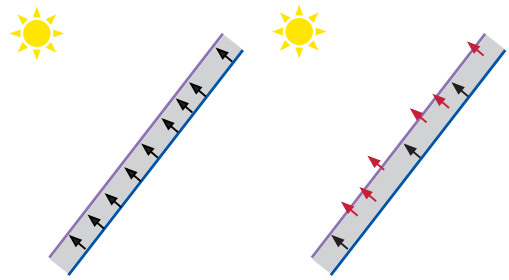
(a) *Raster and ray tracing geometry out of sync while using an exact shadow test*

(b) *Raster and ray tracing geometry out of sync while using an inexact shadow test*

Figure 50-7. *The exact shadow test shows substantial hard shadow artifacts where the geometry used to cast shadow rays does not match the geometry used to test the shadow rays. Applying a stochastic bias to only the pixels known to have this challenge hides the artifact while preserving the overall appearance.*

shader animation fails to exactly match the animation used by the foliage instances in the bounding volume hierarchy (BVH), incorrect self-occlusion is likely to occur. This is easiest to observe with unanimated foliage in the BVH testing against animated foliage in the raster scene. The result is streaky shadows that slice through the foliage as it moves in and out of intersecting with the static representation (Figure 50-7a).

Like most of the other challenges with foliage, a good solution involves leaning into the approximations already occurring. Foliage rendering uses several billboards and leaf cards to represent the volume of leaves on a typical plant. Because the space being rendered can be thought of as a volume, the occlusion testing can be as well. Statistical sampling can approximate shadow results within the volume (Figure 50-7b). This sampling can be accomplished by applying random offsets to the minimum hit distance (`TMin`) for the shadow rays (Figure 50-8). The result is as if a cloud of samples was evaluated above the foliage in the direction of the light source. Clearly, the randomization will result in a noisy shadow result. However, the shadows already rely on denoising and temporal antialiasing passes to produce soft antialiased results. Importantly, placing the bias on `TMin` rather than on the origin for the shadow ray increases the reported distance for the rays that still



(a) *Mismatched geometry* **(b)** *Inexact shadow rays*

Figure 50-8. *The purple and blue surfaces are logically the same, but different ray traced (purple) and raster (blue) representations. Due to the mismatch, the blue surface is in shadow with rays traced directly from the surface. Applying a stochastic offset to the rays allows several samples to avoid the self-occlusion.*

return a shadow result. This aides the denoising process, as the sharpness of denoising is tied to the reported hit distance. See Listing 50-3 for the setup of this inexact shadow test.

Listing 50-3. *Inexact shadow ray setup.*

```
RayDesc Ray = GenerateOcclusionRay(
    LightParameters,
    WorldPosition, WorldNormal,
    RandSample);

// Apply standard bias to avoid depth fighting artifacts.
ApplyCameraRelativeDepthBias(Ray, PixelCoord, DeviceZ, WorldNormal,
     NormalBias);

// If using inexact occlusion tests, apply bias to TMin.
if (NeedsInexactOcclusion())
{
    Ray.TMin += GetRandomOffset() * MaxBiasForInexactGeometry;
}
```

Finally, the inexact shadow testing is only desirable on objects and materials that require the inexact test. Two different solutions have been deployed to solve this. First, as foliage is the primary use case, attributes from the G-buffer such as the shading model are useful to identify geometry wishing to receive the effect. Simply applying it to the `TWO_SIDED_FOLIAGE` shading model in UE4 will cover the most common cases. The downside is that objects like tree branches will not participate. A more exact solution comes from marking the target geometry explicitly. This can be done by an extra bit in the G-buffer when there is room. For the more general case, simply

running an extra stencil-only pass offers a good solution. Because the geometry already exists in the depth buffer, the pass can skip running a pixel shader to produce alpha blending and can rely on setting the depth function to equal for handling pixels cut out by alpha testing.

50.4 SUMMARY

The hybrid translucency and the foliage techniques described in this chapter are insufficient on their own to handle all challenges when adding ray tracing to content authored for rasterization. However, they stand as part of the toolbox of methods to accomplish the task, and when combined with other tools in an engine like UE4, impressive results are possible.

REFERENCES

[1] Amazon Lumberyard. Amazon Lumberyard Bistro. *Open Research Content Archive (ORCA)*, http://developer.nvidia.com/orca/amazon-lumberyard-bistro, 2017.

[2] Carpenter, L. The A-buffer, an antialiased hidden surface method. *ACM SIGGRAPH Computer Graphics*, 18(3):103–108, 1984. DOI: 10.1145/800031.808585.

[3] Everitt, C. Interactive order-independent transparency. http://developer.download.nvidia.com/assets/gamedev/docs/OrderIndependentTransparency.pdf, 2001.

[4] Nature Manufacture. Forest - environment set. *Unreal Engine Marketplace*, https://www.unrealengine.com/marketplace/en-US/product/forest-environment-set, 2019.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.